J. L. Triviño-Rodriguez
and R. Morales-Bueno
Department of Languages and Computer Science
Complejo Tecnológico, Campus de Teatinos
University of M· laga
29071 M· laga, Spain
[trivino,morales]@lcc.uma.es

Using Multiattribute Prediction Suffix Graphs to Predict and Generate Music

Studying the temporal evolution of a parameter in a system is essential to many applications. In these applications, if the value of the parameter at every moment can be represented by a symbol, then the evolution of the system over time can be represented by a sequence of symbols. Statistical modeling of these complex sequences is a fundamental goal of machine learning owing to its wide variety of natural applications, for example statistical models of biological sequences such as DNA (Krogh, Mian, and Haussler 1993).

It is impossible in many applications to determine exactly the next symbol given the previous symbols in the sequence. Thus, learning models try to compute the probability that every symbol must appear in the sequence given the previous sequence of symbols. The set of probabilities of every symbol given the previous sequence of symbols is called the *probability distribution* of the next symbol.

This statistical modeling tries to predict the next symbol in the sequence given the preceding subsequence of symbols—for example, a sequence of musical notes. In this case, statistical modeling will try to predict the next pitch in a melody given the preceding subsequence of pitches. If we consider the empirical probability distribution of the next symbol given a preceding subsequence of some given length, then there exists a length L (the *memory length*) such that the conditional probability distribution does not change substantially if we condition it on preceding subsequences of length greater than L. This feature can be found in many applications related to natural language processing, such as speech recognition (Jelinek 1990; Nadas 1984) and speech tagging (Brill 1994; Merialdo 1994).

Computer Music Journal, 25:3, pp. 62–79, Fall 2001
© 2001 Massachusetts Institute of Technology.

Markov chains (Shannon 1951) model this statistical property and have been applied to model data sequences. The last L symbols of a sequence are called the *state* of the Markov chain. If the parameter of the system can take m different values, and the last L symbols are considered to determine the state of the system, then the system will have m^L different states. To train a Markov chain, the next symbol probability distribution must be computed. This task is usually carried out by computing the relative frequency of a symbol given the preceding subsequence of L symbols in a training sample. The number of conditional probabilities that must be computed is $m^L m = m^{L+1}$. This number grows exponentially with its order L, so only lower-order Markov chains can be considered in practical applications.

An improved model of Markov chains called Probabilistic Suffix Automata (PSA) was developed by Dana Ron (Ron 1996; Ron, Singer, and Tishby 1996). Hence, a PSA is a variable-order L Markov chain, meaning that the order—or equivalently, the memory length—is variable. Unlike Markov chains, this model does not grow exponentially with its order, and hence higher-order models can be considered. Moreover, it produces more intuitive descriptions of practical problems.

A simplified version of the PSA model based on the Lempel–Zip algorithm (Weinberger, Lempel, and Ziv 1982), has been applied by Assayag, Dubnov, and Delerue (1999) to music prediction and generation. In this problem, analyzing the pitches of a melody is not enough. To study these kinds of problems, several attributes must be considered (e.g., durations, fermati, etc.); and a model that considers several attributes has been described by Conklin and Witten (1995) and is called the *multiple-viewpoint system*. A multiple-viewpoint system for music modeling is a collection of

independent views of the sequences of musical notes, each of which models a specific type of musical phenomenon. Nevertheless, neither PSAs nor Markov chains model sequences conditioned by parallel sequences.

The only way to apply Markov chains to these kinds of problems is to use the cross product of alphabets of attributes of the problem as the alphabet of the Markov chain. (An *alphabet* is the set of symbols of the attributes of a problem.) This approach has been followed by Assayag, Dubnov, and Delerue (1999), who use the cross product of the alphabets of pitches and duration of every voice as the alphabet of the Markov chain. Others have used this approach as well, including Xenakis (1960) and Koenig (1970).

For example, let us suppose that a one-voice musical melody is to be modeled. We therefore need to model two attributes (two "viewpoints" in Conklin and Witten's notation): the pitch and duration of every note of the voice. In other words, two parallel sequences of symbols must be considered: a sequence of symbols for the pitch attributes and a sequence of symbols for the duration attributes. Let us consider the alphabet of the attribute pitch P = [c, d, e, f, g, a, b] and the alphabet of the attribute duration D = [1, 2, 3, 4, . . . , 16], where the unit of duration is a sixteenth-note. Then, the alphabet of the Markov chain used to model this music will be composed of the cross product ($P \times D$) of the alphabets P and D. In other words, the alphabet of the Markov chain will be a set A=[(c, 1), (c, 2), . . . , (c, 16), (d, 1), (d, 2), . . . , (b, 16)] where every element is composed of a member of P and a member of D. Hence, if n attributes are modeled and the number of symbols for every viewpoint is m, then the number of elements in the cross product of the alphabets is m^n. Therefore, using the cross product of alphabets makes the length of the Markov chain grow exponentially with the number of attributes. Moreover, the cardinality of the alphabet grows significantly. As the cardinality of the alphabet grows, more melodies are needed to compute the model.

Using PSAs to model these kinds of problems involves exponential complexity, too. On the one hand, PSAs have all the problems of Markov chains. On the other, PSAs cannot handle different memory lengths for every attribute needed to describe the model, and not all attributes need the same memory length. However, if the cross product of alphabets of attributes is used, all attributes must have the same memory length.

To avoid these problems, we have proposed a new model called a Multiattribute Prediction Suffix Graph (MPSG). A detailed description of the model is provided by Trivi o and Morales (2001). The present article describes how the MPSG model could be applied to predict and generate music. This model allows us to compute the next symbol probability function for a sequence of data conditioned by previous values of several parallel sequences, just as in Conklin and Witten's model. However, Conklin and Witten's learning model has the drawback of requiring many external parameters that must be provided by the user. The MPSG learning model can compute a stochastic model of a multiple-viewpoint system automatically without any external parameter. This model can learn sequences generated by one attribute if the sequences generated by the rest of attributes are known. Thus, this model allows us to analyze independently the memory length of every attribute needed to know the next symbol probability distribution for a target attribute. Because this model computes every attribute independently, the number of samples used to compute the next symbol probability distribution does not decrease very much over the Markov model.

The MPSG model is based on the Prediction Suffix Tree (PST) model described by Ron (1996). The PST model was originally introduced by Rissanen (1986) and has been used for universal data compression (Weinberger, Lempel, and Ziv 1982), so the MPSG model is related to the idea of a predictive theory of a given language as a transducer of letters into compressed codes. Conklin and Witten's model is based on this idea and on the relationship between musical experience and entropy as described by Meyer (1957).

The MPSG and the other models mentioned try to determine the contextual information needed to predict the next symbol for every attribute. The context information is composed of the shortest

preceding subsequence of symbols that allows computation of the next symbol with the lowest uncertainty. Thus, these models are related to predictive theories. That is, a predictive model of music tries to predict the next symbol in a piece given the preceding subsequence of symbols. Moreover, using a predictive model, new symbols can be generated sequentially given an initial sequence until an entire composition is generated.

Throughout this article, the usual notation for string handling is used. In this notation, the function *suffix*(*s*) denotes the set of suffixes of a string *s*, and the function *length*(*s*) denotes the length of the string. For example, given a string *s*=*abcdac*, the set of suffixes of the string *s* is *suffix*(*s*)= [**e**, *c, ac, dac, cdac, bcdac, abcdac*] where **e** denotes the empty string.

In the next section, we describe the model. We then describe essential elements of music prediction. Finally, we show several experimental results.

MPSG: The Multiattribute Prediction Suffix Graph

This model is a directed graph (with several restrictions in its layout) and a set of labels. The PST described by Ron (1996) is a subclass of this model and can be considered an MPSG with just one attribute. We begin our discussion with the presentation of basic definitions.

Attribute

An *attribute* is a feature of an object, and it is constituted by a non-empty and finite set of values. If every allowed value is represented by a symbol, then the set of values composes an alphabet. We can express sequences of values of an attribute in terms of strings over the alphabet *X*. The usual notation X^* denotes all the strings of length greater than or equal than zero that are composed of symbols of the alphabet *X*. For example, if *X* = *[a, b, c]*, then X^*= *[***e***, a, b, c, aa, ab, ac, ba, bb, bc, aaa, aab, . . .]*, where **e** again denotes the empty string**.** Examples of attributes are given in Table 1.

When an MPSG is applied to music modeling, the relevant attributes are pitch, duration, etcand so on. However, in order to define formally the

Table 1. Examples of attributes

height	=	*[*medium, large, small*]*
X	=	$[x_1, x_2, x_3]$
Y	=	$[y_1, y_2]$
Z	=	$[z_1, z_2, z_3, z_4]$
pitch	=	the set of 128 pitches from the MIDI specification
duration	=	[1, 2, 3, 4, . . . , 16] (where the unit is a sixteenth-note)

MPSG model, the theoretical model will be described in terms of generic attributes denoted by the character *A* and a superscript that identifies the specific attribute (for example, A^3).

State

The next important definition is that of a *state*. Let $n \in \mathbf{N} > 0$ (we denote the set of natural numbers with **N**), and let $A=[A^1, A^2, \ldots, A^n]$ be a set of *n* attributes, then a *state s* over *A* be a tuple of *n* strings. Each string is composed of symbols of a different attribute of *A:* *s* = (a string of symbols of A^1, a string of symbols of A^2, . . . , a string of symbols of A^n*)*. The set of all states *s* over *A* is denoted by S^A:

$$s \text{ in } A^{1^*} \leftrightarrow A^{2^*} \leftrightarrow A^{n^*}$$
$$S^A = A^{1^*} \leftrightarrow A^{2^*} \leftrightarrow A^{n^*}.$$

Thus, a state could be described as a vector with *n* components. Hereafter, we will identify each component by the superscript of its attribute. Different values of an attribute will be identified by subscripts.

We now present an example of states. Given the set of attributes [*X, Y, Z*] from Table 1, two valid states are $r = \langle x_1x_1x_2, \mathbf{e}, z_4z_2 \rangle$ and $s = \langle x_1, y_1y_2y_1, z_1 \rangle$). Similarly, the state <60 61 65 60, 84> is a valid state for the attributes pitch and duration.

A state in an MPSG is formed by the last symbols in the sample, that is, the last values of every attribute. Thus, a state is the information needed to determine the next symbol probability distribution in a sequence. The state $\mathbf{E} = \langle \mathbf{e}^1, \mathbf{e}^2, \ldots, \mathbf{e}^n \rangle$ com-

posed of n empty strings is always in the MPSG. This state expresses the next symbol probability distribution when the context of the symbols is not considered. The function γ associated with this state is the stationary probability distribution of an MPSG. The stationary probability distribution of an MPSG is the next symbol probability distribution reached by M after time t as t grows to infinity and the context is not taken into account.

For example, given the next sequence of values of the attributes pitch and duration in a Bach chorale shown in Table 2, examples of valid states at time 5 are $<\mathbf{e}, \mathbf{e}>$, $<67, \mathbf{e}>$, $<67, 4>$, $<\mathbf{e}, 4>$, $<67\ 69\ 71, 42>$, etc. The state $<\mathbf{e}, \mathbf{e}>$ expresses the probability distribution of the next symbol when the context of the melody is not considered. The state $<67, \mathbf{e}>$ expresses the next symbol probability distribution when the last pitch in the melody has been 67 and the duration of that pitch is not considered, and so on.

Table 2. A sequence of values of the attributes "pitch" and "duration" in one voice of a chorale by J. S. Bach

Time*	0	1	2	3	4	5	6
Pitch	67	67	74	71	69	67	67
Duration	4	8	4	6	2	4	6

Notes: *"Time" here is simply an integer tag, incremented with each note to indicate the order of events. Its values are not intended to denote equally spaced onset times.

Suffix State

A state s' is a *suffix state* of a state s if every component in s' is a suffix (not necessarily proper) of its respective component in s:

$$suffixst: S^A \rightarrow \wp(S^A)$$
$$suffixst(s) = [s' \mid \text{for all } 0 < i < n, s'^i \text{ in } suffix \mid (s^i)].$$

That is, *suffixst(s)* is the set of all states s' such that for every component i $(0 < I < n)$ of s, the ith component of s' is a suffix of its respective component in s.

As an example of a suffix state, let $s = <x_1x_1x_2, \mathbf{e}, z_4z_2>$ be a state over the set of attributes $[X, Y, Z]$ of Table 1. It follows that $suffixst(s) = [<\mathbf{e}, \mathbf{e}, \mathbf{e}>, <x_2, \mathbf{e}, \mathbf{e}>, <\mathbf{e}, \mathbf{e}, z_2>, <x_1x_2, \mathbf{e}, \mathbf{e}>, <x_1x_2, \mathbf{e}, z_2>, \ldots, <x_1x_1x_2, \mathbf{e}, z_4z_2>]$.

Expanded Attribute

Given the state s over a set of attributes A, the *expanded attribute* of the state s is the attribute with the lowest index i such that all attributes with an index greater than i are the string **e**. This index is computed by the following function:

$$xp: S^A \rightarrow \mathbf{N}$$
$$xp(s) = \min[i \mid 0 < i \leq n \text{ and } (\text{for all } j > i, s^i = \mathbf{e})].$$

The notation $xp: S^A \rightarrow \mathbf{N}$ is used to define the domain and the range of a function. It expresses that the function xp receives as input a value of the set S^A and it returns a natural number.

Informally, a state is the context within which the next symbol probability distribution function is computed. If a state is not large enough to determine this probability distribution correctly, then the state must be expanded; that is, the size of the contextual information stored in the state must be increased. Expansion means increasing the memory length by going back further into the past. The MPSG learning algorithm increases the memory length of a state progressively (one attribute every time) in an ordered way from the first attribute. The function xp (expanded attribute) that is applied to a state s computes the index of the attribute of the state s that was last expanded. It thus represents the set of attributes from the state that could be expanded. In other words, if a state was last expanded in attribute Y, the learning algorithm of MPSGs assumes that the previous attributes to Y have been sufficiently expanded, and so only attributes greater or equal to Y could be expanded.

An short example will illustrate an expanded attribute. Given the state $s = <x_1x_1, y_2, \mathbf{e}>$, the expanded attribute of this state is attribute 2, because attribute 1 has been sufficiently expanded to determine the next symbol probability distribution; otherwise, attribute 1 must be expanded before expanding attribute 2.

Direct Parent

Given a state s over a set of attributes A, the node s' is the *direct parent* of s if s' has the same components as s except for the string $s^{xp(s)}$. This state is computed by the following:

$$ad : S^A - \{\mathbf{E}\} \to S^A \quad b \cdot s'^{xp(s)} = s^{xp(s)}, b \text{ in } A^{xp(s)}$$

$$\text{if } b \cdot s'^{xp(s)}, b \text{ in } A^{xp(s)} \text{ and } s'^i = s^i \text{ for all } i \neq xp(s), \text{then } ad(s) = s'.$$

There is only one symbol b that satisfies this condition. Informally, the function *ad* computes for every state s of S^A, except for the state **E**, a state s' that is identical to s except for the string associated with the expansion symbol ($xp(s)$) denoted by $s^{xp(s)}$. This string is the same string of the state s without the first symbol b.

The direct parent of a state s is the only state s' that could be expanded directly (through the expansion attribute of s' or greater) to obtain s. Therefore, s' is the only state that can be the parent of s in an MPSG. The state **E** has no direct parent because there is no state that can be expanded to obtain a state composed of n empty strings.

In Table 3, an example of several states and their direct parents are shown.

To compute the direct parent (and the expanded attribute) of the state $<x_1x_2, \mathbf{e}, z_1>$, following the definition given above, we might think that this state has no expanded attribute because there is not an attribute j greater than 3 that was the empty string **e**. However, if we consider that there is not an attribute j greater than 3, then all the attributes greater than 3 (there are none) are the empty string.

Expansion Symbol

Given a state s over a set of attributes A, the *expansion symbol* of s is the symbol b in $A^{xp(s)}$ that must be added to $ad(s)^{xp(s)}$, such that $ad(s)$ will be equal to s:

$$xs: S^A - \{\mathbf{E}\} \to \cup_{i=1}^{n} A^i$$

$$xs(s) = b \text{ in } A^{xp(s)} \text{ if } b \cdot ad(s)^{xp(s)} = s^{xp(s)}.$$

Table 3. Example states and their direct parents

State	*Direct Parent*
$<x_1, y_2y_1, \boldsymbol{e}>$	$<x_1, y_1, \boldsymbol{e}>$
$<x_1x_1x_2, y_2, \boldsymbol{e}>$	$<x_1x_1x_2, \boldsymbol{e}, \boldsymbol{e}>$
$<x_1x_2, \boldsymbol{e}, z_1>$	$<x_1x_2, \boldsymbol{e}, \boldsymbol{e}>$
$<x_1, \boldsymbol{e}, \boldsymbol{e}>$	$<\boldsymbol{e}, \boldsymbol{e}, \boldsymbol{e}>$

Table 4. Several example states and their expansion symbols

State	*Direct Parent*	*Expansion Symbol*
$<x_1, y_2y_1, \boldsymbol{e}>$	$<x_1, y_1, \boldsymbol{e}>$	y_2
$<x_1x_1x_2, y_2, \boldsymbol{e}>$	$<x_1x_1x_2, \boldsymbol{e}, \boldsymbol{e}>$	y_2
$<x_1x_2, \boldsymbol{e}, z_1>$	$<x_1x_2, \boldsymbol{e}, \boldsymbol{e}>$	z_1
$<x_1, \boldsymbol{e}, \boldsymbol{e}>$	$<\boldsymbol{e}, \boldsymbol{e}, \boldsymbol{e}>$	x_1

A state here is the context required to compute the next symbol probability distribution. If a state is not large enough to determine this probability distribution correctly, then the state must be expanded; that is, the context information stored in the state must be increased. The expansion symbol of a state is the information added to a state when it is expanded from its direct parent.

As an example, several states and their expansion symbols are shown in Table 4.

Indirect Parent

Given a state s over a set of attributes A, the set of *indirect parents* of s is the set of states s' such that s' is identical to s for all its components except for the string $j < xp(s)$ where $s'^j = suffix\ (s^j)$ is satisfied:

$$ai : S^A \to \wp(S^A)$$

$$ai(s) = \left\{ s' \text{ exists } j < xp(s), \text{ exists } b \text{ in } A^j, b \cdot s'^j = s^j \text{ and (for all } i \neq j, s'^i = s^i) \right\}.$$

Informally, the function *ai* computes the set of states such that every state s' in this set has the same strings associated with every attribute ex-

Table 5. Example states and their direct suffixes

State	*Direct Suffixes*					
$\langle x_2x_1x_1, y_3y_2, \mathbf{e}\rangle$	$\langle x_2x_1x_1, y_3y_2, \mathbf{e}\rangle$,	$\langle x_2x_1x_1, y_2, \mathbf{e}\rangle$	$\langle x_2x_1x_1, \mathbf{e}, \mathbf{e}\rangle$,	$\langle x_1x_1, \mathbf{e}, \mathbf{e}\rangle$,	$\langle x_1, \mathbf{e}, \mathbf{e}\rangle$,	$\langle \mathbf{e}, \mathbf{e}, \mathbf{e}\rangle$
<CGC, 8 8 6>,	<CGC, 8 6>,	<CGC, 6>,	<CGC, **e**>,	<GC, **e**>,	<C, **e**>,	<**e**, **e**>
$\langle x_1x_2, \mathbf{e}\rangle$	$\langle x_1x_2, \mathbf{e}\rangle$, $\langle x_2, \mathbf{e}\rangle$, $\langle \mathbf{e}, \mathbf{e}\rangle$					

cept for an attribute *j*. The attribute *j* is lower than the expansion attribute. The string associated with the attribute *j* of the state *s′* is the same string of the state *s* without the first symbol *b*.

Simultaneous States

Two states *s* and *s′* are considered *simultaneous states* if their defined symbols are the same for both states. That is, *s* is simultaneous to *s*' if and only if there exists a *t* in S^A, *s* in *suffixst(t)*, and *s*' in *suffixst(t)*.

Informally, *s* is simultaneous to *s′* if and only if there exists a state *t* such that *s* and *s′* are suffixes of *t*.

Direct Suffix State

Given a set of attributes *A*, a state *s′* over *A* is a *direct suffix state* of a state *s* over *A* if every component *i* of *s′* previous to the expanded attribute *xp*(*s′*) of *s′* is the same as the component *i* of *s*, and its component *xp*(*s′*) is a suffix of the component *xp*(*s′*) of *s*. That is,

$$suffixd : S^A \to \wp(S^A)$$
$$suffixd(s) = \left\{ s' \text{ (for all } i < xp(s'), s^i = s'^i \text{) and } (s'^{xp(s')} \text{ in } suffix(s^{xp(s')})) \right\}.$$

As an example, several states and their direct suffixes are shown in Table 5.

Multiattribute Prediction Suffix Graph

A common data structure in computer science is the *directed graph*. A graph is a finite set of dots called *vertices* (or *nodes*) connected by links called *edges* (or *arcs*). More formally, a *directed graph* is a set (usually finite) of vertices *V* and set of ordered pairs of distinct elements of *V* called edges. A pair *(x, y)* represents an edge from the node *x* to the node *y* in the graph.

A Multiattribute Prediction Suffix Graph (MPSG) is a directed graph *G* in which each node in the graph is labeled with a state of *G*. These states represent the memory that determines the next symbol probability function. The node labeled with *n* empty strings (where *n* is the number of attributes) is always in the graph. (This node expresses the probability distribution of the next symbol independently of the context.) Nodes are joined by edges labeled with pairs *(, i)*, where *A* and *i* **N**. These pairs allow us to find efficiently every state in the graph. Edges represent how the memory length of the attributes is increased to determine the next symbol probability distribution. Thus, edges represent how the states are expanded while the graph is computed. There are three kinds of edges. (Examples of each will be given shortly, after the MPSG is formally defined.)

Expansive Edges

Let *s* and *s′* be two states. Let us consider that *s* is the direct parent of *s′*. Let us also consider that *s′* gives us more information than *s* in order to determine the next symbol probability distribution. Two probability distributions differ substantially if the ratio between distributions is greater than a lower bound. (This bound will be defined later when the learning algorithm is described.) Then, an edge from *s* to *s′* is added to the graph. This edge is called an *expansive edge*, because the edge increases the memory length of an attribute.

Compression Edges

Let us consider that s and s' are the same states that were mentioned in the case of expansive edges, with the same features. Let us now suppose that there exists a state t that is an indirect parent of s' and that t determines the same next symbol probability distribution that s' does. Then, an edge from s to t is added to the graph. This edge is called a *compression edge*, because the same next symbol probability distribution is determined with shorter memory length.

Backward Edges

Let us suppose that both s and s' determine the same next symbol probability distribution. Let us consider that s and s' have the same memory length, and that s' has more attributes than s with shorter memory length. Then, an edge from the largest direct suffix of s to s' is added to the graph. This edge is called a *backward edge*, because s' is at the left of s in the layout of the graph.

Formal Definition of an MPSG

Let $A = [A^1, \ldots, A^n]$ be a set of attributes, each one composed of a set of symbols $A^i = [A^i_1, \ldots, A^i_{mi}]$ where m_i is the number of symbols in attribute A^i, $1 \leq i \leq n$. Then an MPSG G^x over A for the attribute A^x is a directed graph that satisfies eight requirements. First, every node is labeled with a pair (s, γ_s), where S *in* S^A is a state of G^x, and $\gamma_s{:}A^x[0,1]$ is the next symbol probability function associated with s. This function must satisfy the following equation:

$$\text{for all } s \text{ in } G^x, \sum_{a^x \text{ in } A^x} \gamma_s(a^x) = 1.$$

Another requirement of the MPSG is that the node **E** labeled with a n empty string is always in G^x. Also, if a node s is in G^x, then its direct parent $s' = ad(s)$ is in G^x.

Every edge of an MPSG is labeled by a pair (b, r) where B in A^i, $1 \leq i \leq n$ is a symbol of an attribute of A, and R in $\mathbf{N}, 1 \leq r \leq L$ is a number in $\mathbf{N}$. L is the maximum memory length of the MPSG. From every node, there is at most one edge for each symbol of each attribute of A. Additionally, if a node s and its direct parent $s' = ad(s)$ are in G^x, then there is an expansive edge from s' to s labeled with the expansion symbol $b = xs(s)$ of s, and the number $r = length(s^{xp(s)})$.

Let s and s' be two nodes in G^x. A compression edge labeled with a pair (b, r) could join the node s' to s if and only if there exists a state t that is not in G^x and the following conditions are satisfied:

1. $s' = ad(t)$
2. There exists t', t' in $ai(t)$, and s in $suffixd(t')$
3. There is no t'' in $suffixd(t')$ such that $t'' \neq s$ and t'' in G^x and s in $suffixd(t'')$
4. $r = length(t^{xp(t)})$
5. $b = xs(t)$

The second condition implies that s is a state at the right of s' in the layout of the MPSG, and the third condition implies that there does not exist a state t'' in the MPSG greater than s that can be joined with s' to form a compression edge.

The last requirement of an MPSG is now described. Let s and s' be two nodes in G^x. A backward edge labeled with a pair (b, r) could join the node s' to s if and only if there exists a state t that is not in G^x and the following conditions are satisfied:

1. s is in $suffixd(t)$
2. There is no t'' in $suffixd(t)$ such that $t'' \neq s$ and t'' in G^x and s in $suffixd(t'')$
3. There exists a t', t' in $ai(t)$, and s' in $suffixd(t')$
4. There is no t'' in $suffixd(t')$ such that t'' s' and t'' in G^x and s' in $suffixd(t'')$
5. $r = length(t^{xp(t)})$
6. $b = xs(t)$

Like the conditions of compression edges, the second condition implies that s' is a state at the right of s in the layout of the MPSG. The third condition implies that there does not exist a state t' that is greater than s', and the backward edge can start from t'.

Figure 1 shows an example of an MPSG. This MPSG is composed of the nodes and edges given in Table 6.

Figure 1. An MPSG. Dashed edges denote compression edges, and dotted edges denote backward edges.

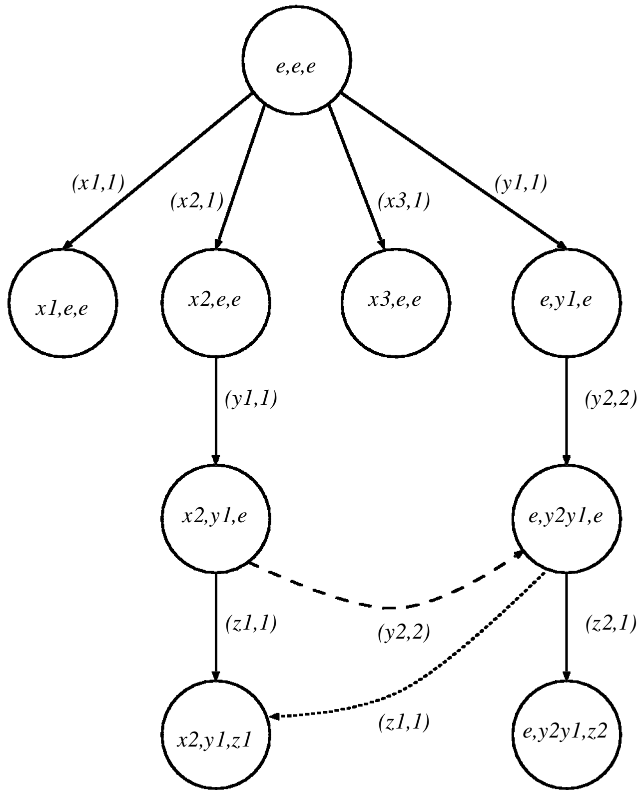


Table 6. Nodes and edges illustrated in Figure 1

nodes = [$\langle \mathbf{e}, \mathbf{e}, e\rangle$, $\langle x_1, \mathbf{e}, \mathbf{e}\rangle$,
$\langle x_2, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_3, \mathbf{e}, \mathbf{e}\rangle$,
$\langle \mathbf{e}, y_1, \mathbf{e}\rangle$, $\langle x_1x_2, \mathbf{e}, \mathbf{e}\rangle$,
$\langle x_2, y_1, \mathbf{e}\rangle$, $\langle \mathbf{e}, y_1y_1, \mathbf{e}\rangle$,
$\langle x_2, y_1, z_1\rangle$, $\langle \mathbf{e}, y_2, y_1, \mathbf{e}\rangle$,
$\langle \mathbf{e}, y_2, y_1, z_2\rangle$]

expansive edges = [($\langle \mathbf{e}, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_1, \mathbf{e}, \mathbf{e}\rangle$, x_1, 1),
($\langle \mathbf{e}, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_2, \mathbf{e}, \mathbf{e}\rangle$, x_2, 1),
($\langle \mathbf{e}, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_3, \mathbf{e}, \mathbf{e}\rangle$, x_3, 1),
($\langle \mathbf{e}, \mathbf{e}, \mathbf{e}\rangle$, $\langle \mathbf{e}, y_1, \mathbf{e}\rangle$, y_1, 1),
($\langle x_2, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_1x_2, \mathbf{e}, e\rangle$, x_1, 2),
($\langle x_2, \mathbf{e}, \mathbf{e}\rangle$, $\langle x_2, y_1, \mathbf{e}\rangle$, y_1, 1),
($\langle \mathbf{e}, y_1, \mathbf{e}\rangle$, $\langle \mathbf{e}, y_1y_1, \mathbf{e}\rangle$, y_1, 2),
($\langle x_2, y_1, \mathbf{e}\rangle$, $\langle x_2, y_1, z_1\rangle$, z_1, 1),
($\langle \mathbf{e}, y_1, \mathbf{e}\rangle$, $\langle e, y_2y_1, \mathbf{e}\rangle$, y_2, 2),
($\langle \mathbf{e}, y_2y_1, \mathbf{e}\rangle$, $\langle \mathbf{e}, y_2y_1, z_2\rangle$, z_2, 1)]

compression edges = [($\langle x_2, y_1, \mathbf{e}\rangle$, $\langle \mathbf{e}, y_2y_1, \mathbf{e}\rangle$, y_2, 2)]

backward edges = [($\langle \mathbf{e}, y_2\ y_1, \mathbf{e}\rangle$, $\langle x_2\ y_1, z_1\rangle$, z_1, 1)]

MPSG Learning Model

The described learning model is similar in spirit to the learning algorithm of PSTs developed by Ron (1996) that is itself based on the model described by Kearns et al. (1994) and Laird and Saul (1994). The MPSG learning algorithm differs, however, from that proposed by Ron et al. (1996) in that it follows a bottom-up approach instead of a top-down approach.

The learning algorithm starts by putting in S all states of length L that have significant counts. Then, states are trimmed by comparing the prediction probabilities of each node to its direct parent. (*Trimming* a state is the inverse of expanding it.) The MPSG must be defined in a top-down approach. In this approach, states are computed by expanding previous states. However, the learning algorithm only can compute the MPSG efficiently in a bottom-up approach. In a bottom-up approach, a state s is trimmed by computing the state s' that will be expanded to s in a top-down approach.

A state is trimmed based upon its empirical probabilities. In this way, a state is not trimmed when the transition probabilities or the observation probabilities of this node differ substantially from its parents. (The ratio between the next symbol probabilities in the direct parent and the next symbol probabilities in the state is smaller than a bound. This bound is defined in the MPSG learning algorithm.)

When the set of states that compose the MPSG is computed, these states must be linked to allow the most representative node of a state to be computed efficiently. To carry out this task, every state is driven along the path of its most representative node. Where there is evidence that the path must be branched, a compression edge or a back-

ward edge is added to the MPSG. (The path must be branched if the state has a symbol that is not in any child of the branched node.)

The MPSG learning algorithm calls several algorithms that are described in Appendix A. This algorithm computes an MPSG in three steps. First, the algorithm `ComputeNodes` computes all the nodes with minimum length that represent the context information needed to compute the next symbol probability distribution of the sample. Second, the nodes and their direct parents are arranged in a tree structure where the algorithm `ConnectNodes` adds the compression and backward edges needed to find a node efficiently. Third, for every node s, the next symbol probability function γ_s is computed.

This algorithm takes eight input parameters:

1. A set of $n > 0$ attributes A
2. The maximum length of the memory L
3. The upper bound of the number of states in the target MPSG q
4. The target attribute x in A
5. An approximation parameter $0 < \varepsilon < 1$
6. Empirical probabilities computed from the training sample $\tilde{P} : S^A \to [0,1]$
7. Empirical probabilities computed from the training sample $\tilde{P} : S^A \times A^x \to [0,1]$
8. A confidence parameter $0 < \delta < 1$

The output value of this algorithm is the MPSG hypothesis $\overline{G}^x$. Pseudocode for the algorithm is given below.

```
N = ComputeNodes()
Gx = an MPSG consisting in a single root
    node labeled by E for all s in N do
  Add state s and all direct parents in
  path to E to Gx
end for
Ḡx = ConnectNodes( Gx, N, E, 0, 0 )
for all node s in Ḡx do
      for all symbol ax in Ax do
```

$$\bar{\gamma}_s\left(a^x\right) = \bar{P}\left(a^x \middle| s\right) * \left(1 - \left|A^x\right| * \gamma_{min}\right) + \gamma_{min}$$

```
      end for
end for
```

This algorithm uses some internal parameters (just as the algorithm proposed by Ron used) when it learns an MPSG. These parameters are described in Appendix A.

Learning Multiple-Viewpoint Systems with MPSGs

A multiple-viewpoint system is described by Conklin and Witten (1995). They call every attribute of the model a *viewpoint*. Every viewpoint models a *type* τ (or abstract property) of events in the sequence. Thus, a viewpoint comprises a partial function $\psi_{\tau}: \xi \to [\tau]$ and a model to predict the next symbol in sequences in $[\tau]^*$, where ξ is the set of valid events and $[\tau]$ denotes the set of all syntactically valid elements of type τ.

An MPSG could be used as the prediction model of sequences in a attribute. In this way, the set of all types τ in the multiple viewpoint system corresponds to the set of attributes A of the MPSG and, for every type τ, $[\tau]$ defines the set of symbols of the equivalent attribute. Thus, an event could be described as a vector with n components, each one formed by a symbol of the alphabet of a different attribute.

For example, *pitch* and *duration* are two attributes that could be considered in order to model musical events. In this example, *pitch* is the pitch of the event represented as an integer value in the range from, say, 60 (C4 or middle C) to 79 (G5, 19 semitones above middle C), according to the MIDI standard. The attribute *duration* is the duration of a event measured in sixteenth-note increments. We need two MPSGs to model these attributes: one MPSG for each attribute. Both MPSGs must have two attributes (pitch, duration), because every MPSG computes the next symbol probability distribution of one attribute conditioned by all the attributes of the problem. The attributes and the alphabets for these two MPSGs are then given by

$$A = [\text{pitch, duration}]$$

$$\textit{pitch} = [60, 61, 62, \ldots, 74, 75]$$

$$\textit{duration} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].$$

Two main viewpoints have now been used in our system to model music: *pitch* and *duration.* Of course, many other viewpoints also exist that could be used in music prediction. Some of these viewpoints have been described by Conklin and Witten (1995) and could be used directly by the MPSG model. Examples of these viewpoints are *contour* (difference between the pitch of a note and that of the preceding note) and *posinbar* (position of the event in the bar).

However, one type of viewpoint cannot be modeled by MPSGs: those that represent musical features with an infinitely large set of values. An MPSG needs a finite set of symbols for every attribute, so an attribute with an infinite number of values cannot be modeled by an MPSG. An example of this kind of attribute includes the start time of an event.

The MPSG model can generate sequences of any length (including infinite length). In a model that represents sequences of infinite length, an attribute whose value always grows, e.g., the start time, must have an infinite number of values. Moreover, if a strictly ordered sequence of values is modeled, the model is bound to reproduce the original exactly. Several derived attributes could be used instead of the start time of an event. For example, instead of using the start time of the event directly, one could measure the difference between the start time of an event and the start time of its predecessor, or the difference in time between the start time of an event and the end time of its predecessor.

Experimental Results

A machine learning system for music generation has been implemented. This system is composed of three MPSGs that model three different viewpoints: *pitch, duration,* and *key signature*. This system has been trained with one 100 Bach chorales from *The 371 Chorales of Johann Sebastian Bach* (edited by Mainous and Ottman 1966). These chorales were also used by Conklin and Witten (1995) for music prediction/generation. The data files used by Conklin and Witten comprised only one voice (the melody) of each of the original chorales, so this system has not been tested with polyphonic music. However, the model could represent several voices in polyphonic music modeling by adding an attribute for every viewpoint of every voice (for example, an attribute for the pitch of the first voice and another attribute for the pitch of the second voice). The main problem of learning polyphonic music is that events in this kind of music are asynchronous (i.e., overlap in time) and MPSGs can only represent synchronous events (i.e., all attributes change their values at the same time). This could be solved by considering elapsed time between events as described by Assayag, Dubnov, and Delerue (1999).

The learning task generated two MPSGs. The number of states of the MPSGs for *pitch* and *duration* are shown in Table 7. In this table, we can see that the memory length needed to represent the Bach chorales is $L = 1$ for both attributes (there are no leaf nodes with length 2 for either attribute), even though the maximum memory length has been set to 4.

These MPSGs have been used in the implementation of a random sequence generator (described below). This program generates a random sequence of events of a given length from the MPSGs that follows the probability distribution described by the MPSGs. The sequences of events are converted to MIDI format and can be played and auditioned. An online demonstration of this task is available at http://alcor.lcc.uma.es/~trivino/musica.

The Sequence Generator Algorithm calls the Most Representative Node Algorithm described in Appendix A. (Essentially, the Most Representative Node Algorithm computes the state (node) that defines the next symbol ($P \times D$) probability distribution for every attribute given the last L symbols of every attribute in a sequence.) The Sequence Generator Algorithm computes n new symbols (a symbol for every attribute) following these probability distributions. Next, these new symbols are added at the end of the generated sequence. This process is repeated until a sequence of length l (given in the input) is generated. The new states needed in each step to compute the most representative node is computed by trimming the last L symbols of every attribute in the generated sequence.

Table 7. Memory length of the learned MPSGs

	MPSG for Attribute Pitch	*MPSG for Attribute Duration*
Total number of nodes	1269	778
Total number of leaf nodes	861	497
# compression edges	60	92
# leaf nodes with length 0	0	0
# leaf nodes with length 1	36	36
# leaf nodes with length 2	825	461
# leaf nodes with length 3	0	0
# leaf nodes with length 4	0	0
# leaf nodes with length 0 in attribute pitch	36	36
# leaf nodes with length 1 in attribute pitch	825	461
# leaf nodes with length 2 in attribute pitch	0	0
# leaf nodes with length 0 in attribute duration	0	0
# leaf nodes with length 1 in attribute duration	861	497
# leaf nodes with length 2 in attribute duration	0	0

This algorithm requires four input parameters:

1. A set of $n > 0$ attributes A
2. An MPSG G^a for every attribute a in A
3. The maximum length of the memory L
4. The length l of the sequence

The algorithm computes a sequence of n-tuples r. The algorithm itself is outlined in pseudocode below.

```
    r = `empty sequence'
    s = E
    for i=1 to l do
          for all a in A do
    s' = nmr( G^a, s )
    t^a = a symbol x^a in A that has
        been selected following the
        probability distribution
        defined by s' in G^a
    s''= last L symbols of the string
         s^a·t^a
  end for
  s = s''
  Add the n-tuple t at the end of r
end for
```

A random sample of 100 pieces consisting of 70 notes each has been generated with this program. Figure 2 shows the probability distribution of the attribute *pitch* of the Bach chorales and the learned MPSG, respectively. The probability distribution of the attribute "duration" is shown in Figure 3.

A statistical test would show that these distributions follow the same probability distribution. However, this is not enough to prove that the model captures correctly the relationships among different values of an attribute in a sequence. For example, a system that would reproduce the content of the chorales by sorting the pitches (playing all the C notes first, then all the C#s, then all the Ds, etc.) would produce the same pitch distribution as the chorale. Moreover, the distance between the chorale distribution and the simulation would be zero.

Although it is difficult to prove that sequences generated by the MPSG follow the same rules as the Bach chorales, the Kolmogorov–Smirnov test (Hollander and Word 1973) could prove that both distributions follow the same short context distri-

Figure 2. Probability distribution of the attribute pitch. The horizontal axis measures semitones above middle C. The vertical axis measures probability from 0 to 1. (The figure shows only the range from 0 to 0.175.)

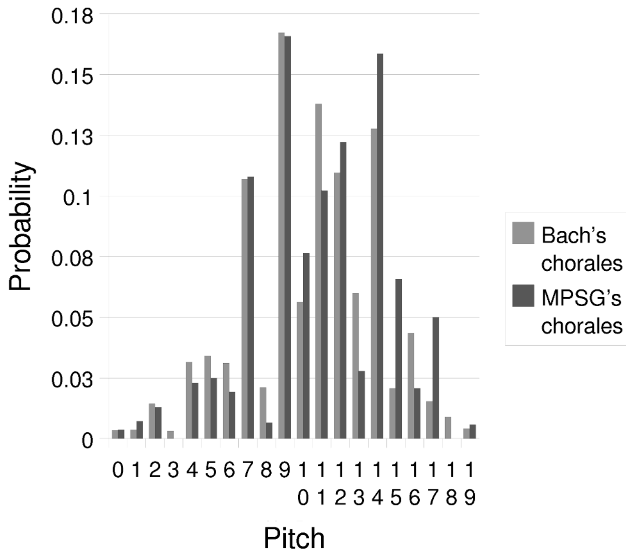
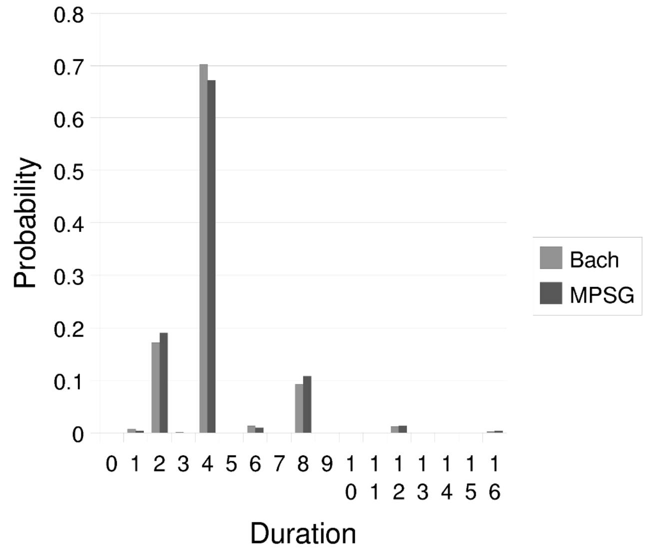


Figure 3. Probability distribution of the attribute duration. The horizontal axis measures sixteenth notes. The vertical axis measures probability from 0 to 1.



bution (the relationship modeled by MPSGs). In order to prove this, the probability distribution of pairs of consecutive pitches in the sequence must been analyzed. Moreover, it is difficult to rearrange the pitches of a sequence maintaining the same probability distribution of pairs of pitches.

Given our alphabet of pitches from C4 to G5, there are 400 possible pairs of two notes. In Figure 4, the probability distribution of the pairs of adjacent notes in Bach's chorales is shown. Clearly, this distribution is very similar to the probability distribution of the pairs of MPSG sequences shown in Figure 5. In both figures, the pairs are arranged first according to the lowest note and second according to the highest note (for pairs that have the same lowest note). Moreover, in both figures, the six most probable pairs are (7, 9); (9, 7); (9, 11); (11, 9); (12, 11); and (14, 12) in terms of semitones above middle C.

Moreover, we applied the Kolmogorov–Smirnov test to test whether the population distribution of these pairs is the same as for the Bach chorales. The maximum value for the function $|F^*_{19}(Bach) - G^*_{19}(MPSG)|$ is 0.025, which is smaller than the critical value ($1.52/\sqrt{400} = 0.076$) for $m = n = 400$ at level $\alpha = 0.20$ for the Kolmogorov–Smirnov test. Thus, the population distribution for pairs of pitches is the same as for the Bach chorales. The same test has been repeated with pairs of three notes. This test has proved that the 8,000 pairs of three notes of the generated music follow the same probability distribution as the pairs of the Bach chorales.

The Kolmogorov–Smirnov test can prove that MPSGs can generate sequences with the same probability distributions and context properties as Bach chorales. However, it cannot show how a human listener perceives the generated music. To evaluate the similarity between the generated music and the Bach chorales, we performed an auditory test where subjects guessed whether fragments they heard came from the original chorale or from the MPSG simulation. Two fragments (one from a Bach chorale and one generated by the MPSG, each approximately 10 measures long) were played in each hearing. Listeners were asked to classify the fragments as either composed by Bach or generated by the MPSG. Fifty-two listeners evaluated the music from the MPSG with this test. They correctly classified the fragments about 55% of the time. This value is almost as low as the 50% required for the machine to pass the Turing test. Moreover, using Hoeffding's bound, the error of this value has been computed to within 12% with a confidence of 0.1. The value 50% is included in the range of 55 ± 12%.

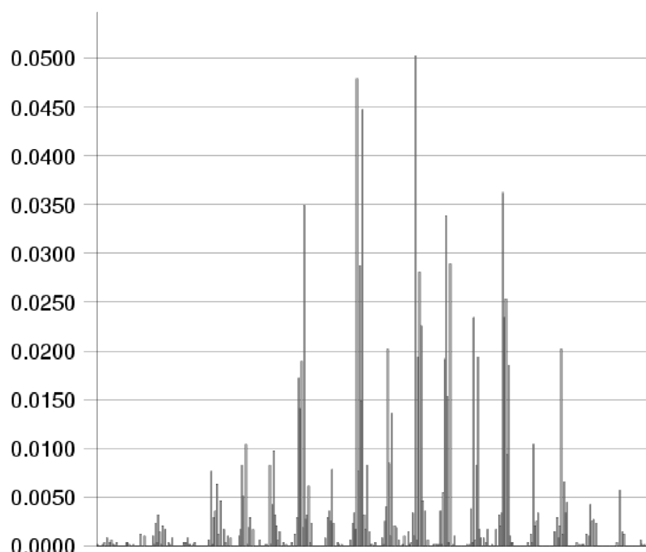


Figure 4. Probability distribution of pairs of pitches of the Bach chorales.

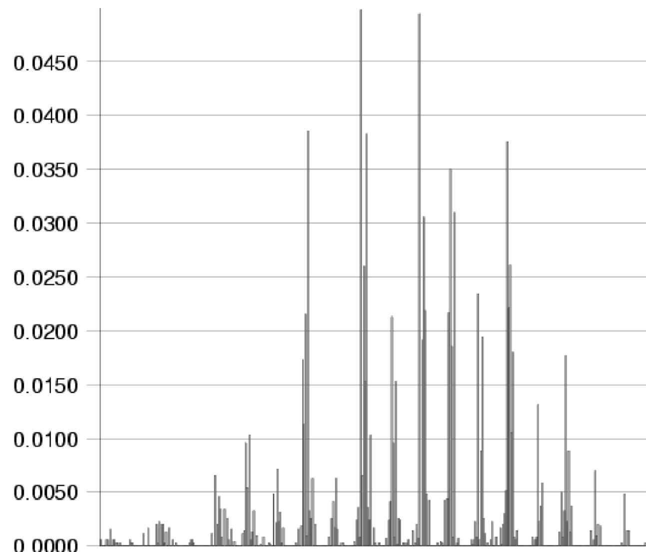


Figure 5. Probability distribution of pairs of pitches in sequences generated by MPSGs.

Conclusion and Future Work

We have presented a new approach to learning generative theories applicable to generating chorale melodies in the style of J. S. Bach. This research is based on the Multiattribute Prediction Suffix Graph learning model that constitutes a new mathematical model for multiattribute, variable-memory-length Markov chains.

Using the MPSG model, only a small number of learning parameters are needed, and new pieces can be generated without heuristic parameters. The quality of the pieces generated is difficult to quantify objectively, but it was shown that the generated pieces have the same probability distribution as the Bach chorale melodies. A demonstration of the music that this model generates is available online at http://alcor.lcc.uma.es/~trivino/musica. Subjects in an auditory test did little better than chance in distinguishing MPSG-generated fragments from Bach chorale melodies. All of the subjects had taken basic courses in music, though a few of them had completed more advanced musical studies.

This work suggests several areas of future research on the prediction and generation of music. The learning model must be extended to consider long memory information. Much of this information could be represented by using attributes of the MPSG in a special way, for example, as time counters or logical switches. Time counters and logical switches are artificial attributes that are added to the problem and computed directly from the other attributes. An example of a time counter is an attribute that represents the start time of the pitch relative to the start time of the bar. Another artificial attribute could be the scale degree (relative distance in semitones from the tonic or first degree of the scale).

The model must also include a more robust treatment of rhythm and meter. The MPSG does not currently account for norms of metrical construction, such as fitting candidate durations to a time signature so that note onsets fall on typical positions within a measure.

Another development might consist of the learning of polyphonic music. The main problem of learning polyphonic music is that events in this kind of music are asynchronous, and MPSGs can only represent synchronous events. This could be solved, as mentioned previously, by considering the elapsed time between events as described by Assayag, Dubnov, and Delerue (1999).

Acknowledgments

The authors would like to thank the anonymous referees selected by *Computer Music Journal* whose comments greatly improved this article.

References

Assayag, G., S. Dubnov, and O. Delerue. 1999. "Guessing the Composer's Mind: Applying Universal Prediction to Musical Style." *Proceedings of the 1999 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 496–499.

Brill, E. 1994. "Some Advances in the Transformation-Based Part of Speech Tagging." *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Menlo Park, California: AAAI Press, pp. 6-–14.

Conklin, D., and I. Witten. 1995. "Multiple Viewpoint Systems for Music Prediction." *Journal of New Music Research* 24(1):51–73.

Hollander, M., and D. A. Wolfe. 1973. *Nonparametric Statistical Methods*. New York: Wiley.

Jelinek, F. 1990. "Self-Organized Language Modeling for Speech Recognition." In A. Waibel and K. F. Lee, eds. *Readings in Speech Recognition*. San Mateo, California: Morgan Kaufmann, pp. 450–503.

Kearns, M., et al. 1994. "On the Learnability of Discrete Distributions." *The 26th Annual ACM Symposium on Theory of Computing*. New York: Association for Computing Machinery, pp. 273–282.

Koenig, G. M. 1970. "Project 1: A Programme for Musical Composition." *Electronic Music Reports* 2:32–44.

Krogh, A., S. I. Mian, and S. I. Haussler. 1993. "A Hidden Markov Model That Finds Genes in E. Coli DNA." Technical Report UCSC-CRL-93-16. Santa Cruz, California: University of California at Santa Cruz.

Laird, P., and R. Saul. 1994. "Discrete Sequence Prediction and Its Applications." *Machine Learning* 15:43–68.

Mainous, F. D., and R. W. Ottman. 1966. *The 371 Chorales of Johann Sebastian Bach*. New York: Holt, Rinehart, and Winston.

Merialdo, B. 1994. "Tagging English Text with a Probabilistic Model." *Computational Linguistics* 20(2):155–171.

Meyer, L. B. 1957. "Meaning in Music and Information Theory." *Journal of Aesthetics and Art Criticism* 15:412–424.

Nadas, A. 1984. "Estimation of Probabilities in the Language Model of the IBM Speech Recognition System." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 32(4):859–861.

Rissanen, J. 1986. "A Universal Data Compression System." *IEEE Transactions on Information Theory* 29(5):656–664.

Ron, D. 1996. "Automata Learning and Its Applications." Ph.D. thesis, Massachusetts Institute of Technology.

Ron, D., Y. Singer, and N. Tishby. 1996. "The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length." *Machine Learning* 25(2–3):117–149.

Shannon, C. E. 1951. "Prediction and Entropy of Printed English." *Bell Systems Technical Journal* 30(1):50–64.

Trivi o, J.L, and R. Morales. 2001. "MPSGs (Multiattribute Prediction Suffix Graphs)." Technical Report ITI-2001-6. M· laga, Spain: University of M· laga.

Weinberger, M. J., A. Lempel, and J. Ziv. 1982. "A Sequential Algorithm for the Universal Coding of Finite-Memory Sources." *IEEE Transactions on Information Theory* 38:1002–1014.

Xenakis, I. 1960. "Elements of Stochastic Music." Gravesaner Blätter 18:84–105.

Appendix A: Subalgorithms of the MPSG Learning Algorithm

Figure A1 shows a diagram that describes how the different algorithms used in the learning task of a MPSG are linked. In the course of the algorithm, we refer to severeal internal parameters which are all simple functions of ε, n, L, A, and q. These internal parameters are given by the following:

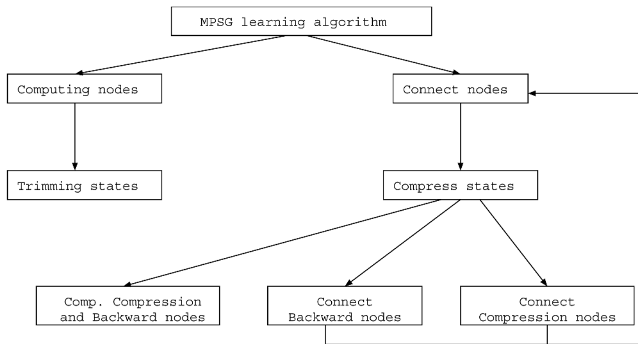
$$\varepsilon_2 = \frac{\varepsilon}{48Ln}$$

$$\gamma_{min} = \frac{\varepsilon_2}{\sum_{i=1}^{n} \left|A^i\right|}$$

$$\varepsilon_0 = \frac{\varepsilon}{2qLn\log\left(\frac{1}{\gamma_{min}}\right)}$$

$$\varepsilon_1 = \frac{\varepsilon_2 \gamma_{min}}{8q\varepsilon_0}.$$

Figure A1.



Algorithm A1: Trimming States

Given a state s, the Trimming States Algorithm searches for a state d in the set of parents of s. The state d must have shorter memory length than s, and it must determine the same probability distribution as s. To carry out this task, this algorithm computes the set of all states that determine the same probability distribution as s for every symbol with a non-negligible probability. Next, for every state d of this set, the algorithm computes the set C of all simultaneous states to d with non-negligible probability. If all the states of the set C determine the same probability distribution as d, then the state d is considered instead of s; otherwise, the state s is added to the set of trimmed states.

This algorithm receives four input parameters:

1. Parameters of the "computing nodes" algorithm
2. The set of states S to trim
3. The state to be trimmed s
4. The set of states trimmed N. (The computed trimmed states when s is trimmed are added to N.)

The algorithm itself is given below in pseudocode.

D = [$y|y$ in P and for all a^x in $A^{x:}$

$$\bar{P}\left(a^x|y\right)?\left(1+\varepsilon_2\right)*\gamma\ min\ \bullet\frac{\bar{P}\left(a^z|y\right)}{\bar{P}\left(a^x|s\right)}\leq 1+3*\varepsilon_2\}$$

```
nnode= T
while D ≠ 0 and nnode do
  d= maxarg_{y in D}(i, y^i ≠ s^i)
    D= D-{d}
    if d in (S ∪ S' ∪ N) then
      nnode= ⊥
    else
      C= {y|y in (S ∪ S' ∪ N) and
      P̄(y) ? (1–ε_1)*ε_0 and
      d is simultaneous with y}
    if for all c in C, d in suffixst
    ( c ) and
        for all a^x in A^x:
```

$$\bar{P}\left(a^x|y\right)?\left(1+\varepsilon_2\right)*\gamma\ min\ \bullet\frac{\bar{P}\left(a^z|y\right)}{\bar{P}\left(a^x|s\right)}\leq 1+3*\varepsilon_2\}$$

```
        then
          S = S ∪  S{d}
                nnode= ⊥
            end if
        end if
    end while
  if nnode then
    N= N ∪ {s}
  end if
```

Algorithm A2: Computing the Nodes

This algorithm follows a bottom-up approach. It begins with the set of all states in the training sample that are progressively trimmed. The states are trimmed in two ways: if a state has a negligible probability, it is replaced for all its parents; otherwise, the state is trimmed with the "trimming state" algorithm.

This algorithm has the same input parameters as the MPSG Learning Algorithm. The algorithm computes the set of states N.

```
S= {y|y is in the sample and (for all
0i<n, length(y^i)=L)}
N= ∅
while S ≠ ∅ do
    S'= S
    S= ∅
    while S'≠ ∅ do
      Pick up any s in S'
      S'= S'-{s}
      if P̄(s) > 0 and s ≠ E then
        P= {y|y in ({ad(s) ∪ ai(s))
```

```
   and y ≠ E}
   if P̄(s)<(1-ε1)*ε0 then
P'={y|y in P and P̄(y)<(1-ε1)*ε0]
P"={y|y in P and P̄(y)?(1-ε1)*ε0]
    and (not exist t in (S[S'[N),
      P̄(t)?(1-ε1)*ε0] and
      y is simultaneous with t}
            s = S ∪ P' ∪ P"
      else
            TrimState( S, N, s )
                  end if
            end if
      end while
end while
```

Algorithm A3: Computing Compression and Backward Nodes

This algorithm classifies the target nodes *r* of the compression and backward edges from a state *c* as nodes that must be linked by a compression edge or a backward edge. This task is carried out by computing the state *z*. If the state *y*, computed by trimming the state *r*, is a suffix state of *z*, then *r* is the target for a compression edge. Otherwise, *r* is the target for a backward edge.

This algorithm takes five input parameters:

1. Target states of the compression and backward edges S"
2. A current state *c*
3. A current attribute *u*
4. A current symbol a^x
5. The current memory length *p*

The algorithm computes two values:

1. A set of compression nodes *C*
2. A set of backward nodes *R*

Pseudocode for the algorithm is given below.

```
z= c
zu= axzu
R= ∅
C= ∅
for all r in S'' do
      y= r
      yu= yup...yu0
      for i= u+1 to n-1 do
            yi= e
      end for
      if y in suffixst( z ) then
            C= C+{r}
      else
            R= R+{r}
      end if
end for
```

Algorithm A4: Connecting Backward Nodes

This algorithm adds all backward edges from state *c* to the states in *R*. The target state of an edge in the graph is computed by trimming every state in *R*. When an edge to a state *r* is added, all the states simultaneous to *r* are removed from *R*. The Connecting Backward Nodes Algorithm takes seven input parameters:

1. Parameters of the Connecting Nodes Algorithm
2. The simultaneous states *S'*
3. A set of backward nodes *R*
4. A current state *c*
5. A current attribute *u*
6. A current symbol a^x
7. The current memory length *p*

This algorithm computes an MPSG G^x.

```
while R ≠ ∅ do
   Pick up any r in R
   ru= rup...ru0
   for i=u+1 to n-1 do
   ri= e
   end for
   E={y|y in R and y is simultaneous
   with r}
   R= R-E
   Add a backward edge from c to r
   labeled with (ax, p+1)
   Gx = ConnectNodes( Gx, ES', r, u, p+1)
end while
```

Algorithm A5: Connecting Compression Nodes

This algorithm computes the largest state *r* in *C*. (Compression states are added in an ordered way from large to short states.) Then, a compression edge from *c* to the trimmed state *r* is added to the graph. The algorithm takes seven input parameters:

1. Parameters of the Conecting Nodes Algorithm
2. The simultaneous states S'
3. A set of compression nodes *C*
4. A current state *c*
5. A current attribute *u*
6. A current symbol a^x
7. The current memory length *p*

The output of this algorithm is the MPSG G^x.

```
if C ≠ ∅ then
    r= maxarg_{y in C}( ∑_{i=0}^{i<u} length(y^i))
    r^u= r^u_p...r^u_0
    for i= u+1 to n-1 do
       r^i= e
    end for
    Add a compression edge from c to r
    labeled (a^x, p+1)
    ^x = ConnectNodes( G^x, C ∪ S', r, u,
    p+1 )
end if
```

Algorithm A6: Compressing States

This algorithm takes seven input parameters:

1. Parameters of the Conecting Nodes Algorithm
2. The simultaneous states *S′*
3. Target states of the compressed and backward edges *S″*
4. The current state *c*
5. A current attribute *u*
6. A current symbol a^x
7. The current memory length *p*

The output of the algorithm is the MPSG G^x.

```
(C,R) =
ComputeCompressionBackwardNodes( s", c,
u, a^x, p)
G^x = ConnectBackwardNodes( G^x, S', R, c,
u, a^x, p)
G^x = ConnectCompressionNodes( G^x, S', C,
c, u, a^x, p)
```

Algorithm A7: Connecting Nodes

This algorithm searches the graph recursively for nodes that can be replaced by a compression edge. When one of these states is found, this algorithm removes this state from the set of states and calls the Compressing States Algorithm to add the compression and backward edges. The Connecting Nodes Algorithm requires six input parameters:

1. Parameters of the MPSG Learning Algorithm
2. The unconnected MPSG G^x
3. A set of states *S*
4. A current state *c* in S^A
5. A current attribute $0 \geq t < n$
6. A current memory $0 \geq p < L$

The output of this algorithm is the connected MPSG G^x.

```
if not is leaf c then
  if p≥L then
    t= t+1
    p= 0
  end if
  for u= t to n-1 do
    S'= {y|y in s and length(y^u)p}
    for all a^x in A^x do
      S''={y|y in S and length(y^u)>p
      and y^i_p=a^x}
      S= S-S''
      if exists an expansive edge from
      c labeled (a^x, p+1) then
          d = node pointed by the
          expansive edge from c
          labeled with (a^x, p+1)
          G^x = ConnectNodes(G^x, S'S",
          d, u, p+1)
```

```
        else if S" ≠ ∅ then
Gx = CompressState(Gx, S', S'', c, u,
ax, p)
        end if
      end for
    end for
  end if
```

Appendix B: Subalgorithms of the Sequence Generator Algorithm

Given a state s and an MPSG G^x over a set of attributes A, the *most representative node* of s in G^x is the node s' in G^x reached once G^x has been traversed in an ordered way for every attribute A^i of s starting from the initial node ***E***. The simplest case occurs when s is in the MPSG. Then, s is the most representative node itself.

This algorithm takes two input parameters: s in S^A, and G^x. The Most Representative Node Algorithm computes the function $s' = nmr(G^x, s)$.

```
  s'= E
for i= 1 to n do
  r= 1
  ti= si
  while ti ≠ e; and there exists an edge
  from s' labeled
  with the right symbol of ti and the
  number r;
  and the state pointed by this edge is
  simultaneous with s do
    s'= s'' where s'' is the node
    pointed by the edge from s' labeled
    with the right symbol of ti and the
    number r; and s'' is simultaneous
    with s
    Remove the right symbol of ti from
    ti
    r= r+1
  end while
end for
```