A Self-Learning Musical Grammar, or "Associative Memory of the Second Kind"

Teuvo Kohonen

Helsinki University of Technology
Laboratory of Computer and Information Science
Rakentajanaukio 2 C, SF-02150 Espoo, Finland

Abstract

A context-sensitive generative grammar that learns its production rules automatically from examples and optimizes the length of context for each individual production rule on the basis of conflicts occurring in the source material is described. It has been applied to the generation of new melodic passages and counterpoint according to a certain style. Music produced by this method generally sounds smooth, continuous, and pleasant.

1. Introduction

The "Composing Machines" have a longer history than generally thought. For instance, the Prague Cistercian M. Vogt (1719) bent nails to different shapes to designate melodic turns, and tossing them in the air, recorded the melodic successions as they lay on the floor.

It is not possible to survey here the development of ideas in computer music. One of the traditional approaches, however, may be mentioned. It is based on Markov processes. Each note (pitch, duration) is thereby regarded as a stochastic state in a succession of states. The probability $Pr = Pr(S_i | S_{i-1}, S_{i-2}, ...)$ for state S_i, on the condition that the previous states are S_{i-1}, S_{i-2}, ..., is recorded from given examples. Usually three predecessor states are enough. New music is generated by starting with a key sequence to which, on the basis of Pr and, say, the three last notes, the most likely successor state is appended. The augmented sequence is used as a new key sequence, and so the process generates melodic successions *ad infinitum*. Auxiliary operations or rules are necessary to make typical forms (structures) of music out of pieces of melodic passages.

In Artificial Intelligence approaches, the music programmers, on the basis of their expertise, usually heuristically construct a number of rules which describe different aspects of music: forms, melodic and rhythmic rules, the counterpoint etc. Melodies and accompanying parts are then automatically produced by generative grammars into which the rules are built (cf., e.g., [1]).

Among the main problems encountered in computer music one may mention the following. First, it is rather difficult to find the start and stop of a possible theme from productions automatically. Second, it is difficult to maintain a certain "style" over a passage and to switch to a new one, which often, especially in classic music, would need a modulation (change of scale). Third, although, e.g., certain interval rules are frequently applied to prevent bad melodic turns, there is no guarantee for a melody constructed on the basis of such rules being beautiful at all.

As I had worked since 1985 with a new type of self-learning grammar termed *Dynamically Expanding Context* [2,3], the idea of applying it to music had several times come to my mind. After many attempts I finally succeeded in generating pieces of melodies which sounded promising. It has to be emphasized that *we are not constructing any rules heuristically*; all the "grammatical" productions are derived from examples, for which we initially took the well-known Inventions of J.S. Bach. The rules of the grammar in fact describe what might be called a *motive* in music; i.e. a rather short elementary sequence, say, three to six successive notes which do not yet form a theme. It turned out that the original motives convey the style and beauty to the results.

Our productions, however, should not be regarded as "scrambled Bach". The special grammar that I am describing below is in fact able to pick up from memory long sequences of motives of the same style, thereby maintaining a high degree of "coherence" in music. This is due to the fact that while this grammar belongs to the category of context-sensitive grammars, the length of context (corresponding to the number of predecessor states in a Markov process) is a variable number; the necessary amount of context is defined dynamically, on the basis of *conflicts* that occur in the examples, and so a very delicate compromise between generality and selectivity of the rules is achievable. The resulting melodies, although not containing long copies of the original pieces, generally sound very smooth and pleasant, and rather faithful to the original style.

There already exist several variants of the grammar. As context, we may use either previous notes, or a couple of previous notes preceded by symbols which describe chords of the previous bars or their parts. The grammar is neither restricted to one-part melodies; it is rather straightforward to edit the second, third, etc. part or voice to the melody, by taking the context from the first part, and generating the notes of the other parts in the same way as new notes to the melody are produced. For practical reasons, and to guarantee the best possible counterpoint, the whole melody is constructed first, and after that the accompanying parts in separate passes.

2. Simple exemplification of the Dynamically Expanding Context method

In general, a grammar describes regularities in a series of events. Consider first a very simple example where the letters describe a sequence of states, e.g., notes:

ABCDEFG ... IKFH ... LEFJ ...

If we would try to deduce the next letter on the basis of *one* letter, say , F (which occurs several times) we would see a threefold *conflict*: the continuation may be G, H, or J. If we would try to increase specificity of letter patterns by taking the symbol in front of F for *context*, we could still see that a twofold conflict prevails: the successor of EF could be G or J. With two symbols in front of F, all the conflicts would be resolved. However, KF would already uniquely define H as its successor (denoted KF $\rightarrow$ H, and meaning that KF is uniquely followed by H or that KF implies H, or that KF generates the *production* H). It would then be superfluous to have a longer context for it (IKF $\rightarrow$ H). Too long context means a too specific and stiff rule.

The basic idea behind the Dynamically Expanding Context is that the optimal length of context, for each symbol (like F) separately, is determined on the basis of examples and in particular, by the conflicts of the above type occurring in them.

To construct the variable-context rules systematically, we first think of a series of hypothetical contexts of gradually increasing length at a particular symbol; e.g., around the first occurrence of F in the previous example we then define the *context levels* by

Level	Context
0	-
1	E
2	DE
3	CDE
...	

up to a certain maximum limit (we had 8 levels). Whenever we start recording production rules, we always start at level 0. (The same is due when we *apply* the rules, say, for the generation of music.) For the first tentative rule, referring to F, we then take F $\rightarrow$ G. Upon scanning the example, we then also find F $\rightarrow$ H, indicating that both of these rules should be invalidated and replaced by EF $\rightarrow$ G and KF $\rightarrow$ H, respectively. At the third place, we again start with F $\rightarrow$ J, and then because there is a conflict with a previous case, continue with EF $\rightarrow$ J; since the conflict persists, we must take for the last rule LEF $\rightarrow$ J, and also update the previous, still conflicting rule to DEF $\rightarrow$ G.

We must not *delete* the invalidated rules, because they are needed both for checking of the conflicts, and to construct the valid rule when generating new productions. Instead, we provide each rule with a validity indicator, a *conflict bit* which is initially 0. When a conflict is encountered, and a rule is invalidated, the conflict bit is changed to 1.

The constructed rules are best stored in memory like entries in the so-called *relational data base*. Each entry is a triplet, consisting of the left part of the production rule (like F), the right part of the production rule (like G), and the conflict bit. *The entries are always searched on the basis of the left part.*

Although updating of all counterparts of the conflicting cases might be done immediately when a conflict is found, it is far simpler if only the last of the conflicting rules is updated, and the previous one (at which the conflict occurred) is tentatively only invalidated by changing its conflict bit to 1. If the examplary data are then scanned iteratively a sufficient number of times (in fact, at maximum half of the number of levels, or four in our case) then it is possible to show that all the earlier counterpart rules will sooner or later become updated, too. To show that this is true for the above example, consider the construction of rules only around symbol F (although all the storing operations must be done for all the other symbols, too). A memory location is the triple (left part, right part, conflict bit).

Example.

Construction of the memory and the grammar (around F only):

Row in memory	**Left part**	**Right part**	**Conflict bit**

(Check first that F was not yet stored in any left-part field. Store the first occurrence of F with level 0 context.)

1	F	G	0

(Next, the second occurrence of F is considered; searching on the basis of F, G is found from memory, indicating a conflict with H. The conflict bit of row 1 must now be changed, and the second occurrence stored with level 1 context on row 2, first checking that it was not yet in memory.)

1	F	G	1
2	KF	H	0

(Next, the third occurrence of F is considered; searching on F, G is found from the memory and also that the conflict bit is already 1, and need not be changed. The third occurrence must thus be stored with level 1 context, first checking that it was not yet stored.)

3	EF	J	0

(Next, iterate over the first occurrence of F. Searching on the basis of F, find conflict bit 1. Expand context, find on the basis of EF that there was still a conflict, namely, J in memory. Change the conflict bit on row 3, and store the first item with level 2 context.)

3	EF	J	1
4	DEF	G	0

(Iterate over the second occurrence of F. On the basis of F, the conflict bit 1 is found. On the basis of KF, no conflict exists any longer. Now continue with the third occurrence of F. On the basis of F, the conflict bit 1 is found. On the basis of EF, the conflict bit 1 is still found. Now LEF is not yet in memory, and is stored.)

5	LEF	J	0

(Further iterations do not cause any changes in memory. We see that the valid rules are left on rows 2, 4, and 5, where the conflict bit is 0.)

Generation of a new symbol to a key sequence:

Let us start with a key sequence, say, CDEF. A search on the basis of F indicates that the conflict bit is 1, and such a rule is invalid. A search on the basis of EF still yields the conflict bit 1. The second-level context, search on the basis of DEF, finds a valid rule on row 4, and the new symbol is thus the production G.

It is no surprise that the new symbol is the same as in one of the original occurrences, from which the memory was formed. If we had preferred a random choice from several alternatives to have more variance in the productions, we might have considered, e.g., level-2 and level-1 productions found in this searching process (not caring about the conflict bit); then we would have had two alternatives, J and G, of which we could have randomly selected either one for the new symbol, still having a certain degree of continuity (context-dependence).

The structure of information stored in memory can be visualized by a graph which interrelates the rules. For each particular symbol (such as F above) there exists a tree, and its root corresponds to context level 0. If there was a conflict in the examples, at least two arcs then emanate from the root and lead to the nodes of level 1. The corresponding productions (right sides), like in formal grammars, can be written at the arcs. The nodes which represent leaves, and at which the respective last productions are also written, correspond to the valid final rules (with conflict bit 0), whereas for all the other nodes the conflict bit is 1, indicating that the respective rule has been invalidated.

Generation of new, partly random sequences can now be illustrated in the following way. Assume that the key sequence again is CDEF, and start at the root F, following the arcs until the corresponding leaf is found. Along the path from the root to the leaf one eventually finds several productions (alternatives for new symbols). By a given *depth parameter* it is possible to define how many nodes backwards from the leaves at maximum one shall take into consideration, to randomly pick up one of their productions for the new symbol (note).

If such a randomizing is used, it may happen that when tracing the tree, a mismatch of the context happens at some level. The same is due if the maximum defined context level is exeeded (due to many conflicts at a particular symbol). This, however, does not cause any serious problems in the present application in which the grammar is only used as an "intelligent random number generator". Starting with the root, the tree is anyway traced up to the last matches, whereafter, according to the depth parameter, the new symbol is picked up from the last nodes below it.

Fig. 2 exemplifies typical context levels in a good-sounding melody. The numbers indicate the levels of context which have resulted at each consecutive production step, each number thereby corresponding to one generated note. Two facts are salient: 1. The context level has indeed a great variance, which is expected to make a big distinction, e.g., with respect to Markov-process music. 2. There are several level-0 productions. This is due, because we used parameter values which were supposed to imply greater random variations. Context level 0 in fact means that only two successive notes at that place have been copied from the original material (which typically contains 2000 or more notes in one grammar or memory).

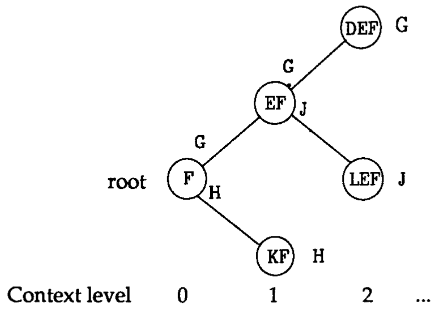


Fig. 1

```
3 2 2 3 3 1 1 2 2 1 2 3 4 2 2 1 2 3 2 1
1 1 1 2 1 2 1 1 3 2 4 1 2 2 4 5 1 2 5 3
3 4 2 1 2 3 4 4 1 1 3 1 1 2 1 2 2 4 2 2
3 4 1 2 2 2 0 2 1 1 1 1 2 2 1 2 2 1 3 0
2 3 4 0 1 2 1 4 3 3 1 1 2 2 4 2 6 3 4 3
2 2 2 2 3 3 2 2 3 3 2 2 3 3 2 2 2 2 3 2
1 6 2 3 2 2 3 3 2 1 1 1 2 2 1 2 2 1 1 1
2 1 1 1 0 2 0 2 0 1 2 2 2 1 2 2 4 4 3 2
2 3 2 1 6 2 3 2 2 2 2 3 3 2 2 2 2 2 2 1
2 2 4 5 1 2 2 1 2 3 2 1 6 2 3 2 5 5 3 1
2 0 2 0 1 2 2 2 2 3 3 2 2 2 2 1 2 2 4 0
1 2 1 4 2 3 2 2 3 2 1 6 3 3 1 1 2 2 4 4
3 2 2 3 3 2 6 5 3 2 2 3 2 1 1 1 1 2 2 2
```

Fig. 2

3. Taking harmonic successions into account

It may be obvious from the above that the "scope" of the basic grammar is still rather short, covering a few successive notes only. It does not deal with any higher forms at all. While, due to the correlation of the melodic segments, the music produced by this method still yields the feeling of longer continuity, the traditions in western music would anyway prefer more regular and "coherent" chord progressions. A modification of the grammar to that end is easy.

We found experimentally that the last two symbols of the key sequence should always consist of absolute notes (pitch, length), whereas for the previous symbols in the grammatical rules describing the productions, one could take symbols of *chords* formed of the melody notes over the preceding bars, half-bars, quarter-bars, etc. Since there may be passing notes in the melody which do not belong to the intended chord, identification of a chord over a short succession of notes must be based on approximate pattern recognition techniques. For instance, one may form a histogram of the various pitches (actually, degrees modulo 12 in the chromatic scale), thereby also taking into account the lengths of the notes, and comparing such a histogram to those formed of the pure chords, to find the best match. Although this method may not yield a hundred per cent sure identification of chords, say, in classic-romantic music, absolute accuracy is neither necessary, since a progression of approximate chords will generally also sound pleasant, and it is still descriptive of the original style.

When the new melody is generated, the chords of its segments can similarly be estimated by this pattern recognition method, yielding the symbols needed in the production rules of the grammar.

4. An example

The two-part music, shown in Fig. 3 has been made with a grammar described in section 3 into which the three-part Inventions ("Symphonies") of J.S. Bach were encoded, and of which only two parts were utilized. All these pieces were first transposed into the C major (or A minor) scale (in which the productions are generated). This is perhaps not the most convincing example musically, since there exist many short copies from most of the original pieces. Nonetheless these copies have been merged in a graceful way. None of the copies extends over a bar. On the other hand, it is very easy to avoid copies of themes even completely, e.g., by diatonic transformation of the source material or of the productions. In this way we have generated many totally new melodies.

5. Why should this method be discussed in the context of Associative Memory?

Certainly the present method does not belong to the pure Neural Network algorithms. It is rather to be regarded as an abstract scheme which exemplifies the *competence* of idealized learning algorithms.

Each production rule, however, is equivalent to an Associative Recall operation. As a matter of fact, the rules were encoded in the program code by a software content-addressing method (hash coding). The left part in the rules assumes the role of a key input, but in contrast to usual Associative Memories, there are *two* outputs from the memory: the production, and the conflict bit. The latter represents some kind of "credibility information". So, if the rule system were stored in an Associative Memory and made accessible on the basis of their left parts, the conflict bit would tell about the production: "That's it", or, "That's not it yet, keep looking". In the latter case, one must assume an external supervision which modifies the range of context at input. From another point of view, this algorithm then describes an architecture, a memory system which is monitored by an "attention control" external to it.

While the formats of input fields in usual Associative Memory schemes are fixed, the above "attention control" now introduces an elementary system organization or control architecture to the system which varies the input format. This feature might be regarded as a paragon to further developments in Associative Memory architectures. For this reason, I have also nicknamed the above principle *"Associative Memory of the Second Kind"*.

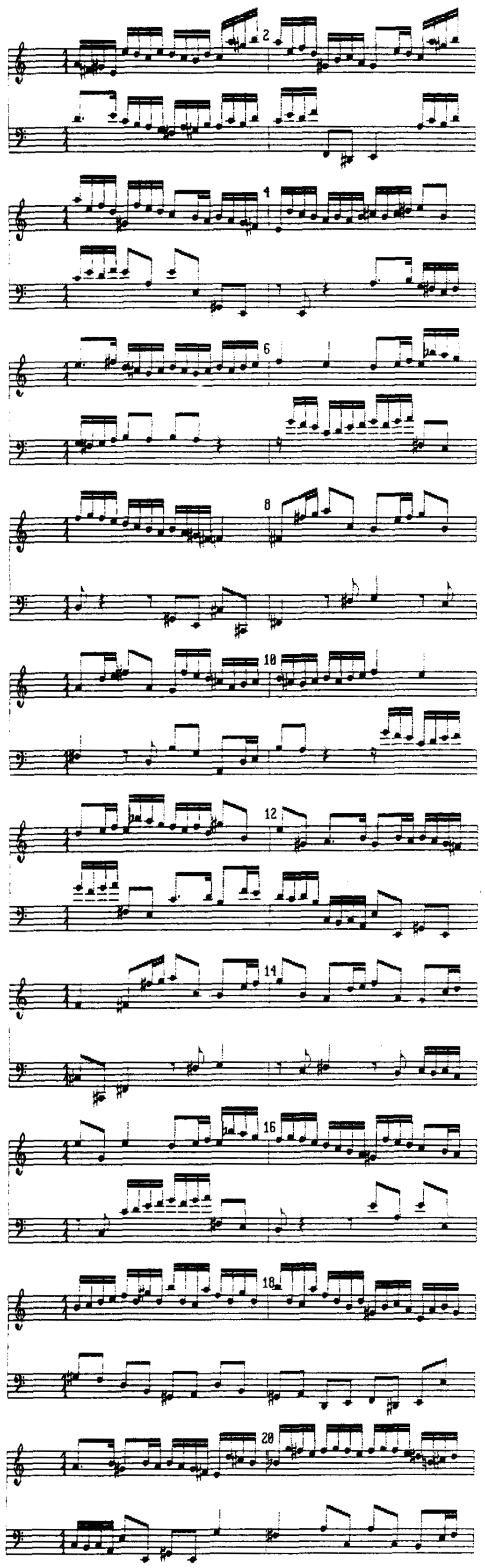
Acknowledgement.

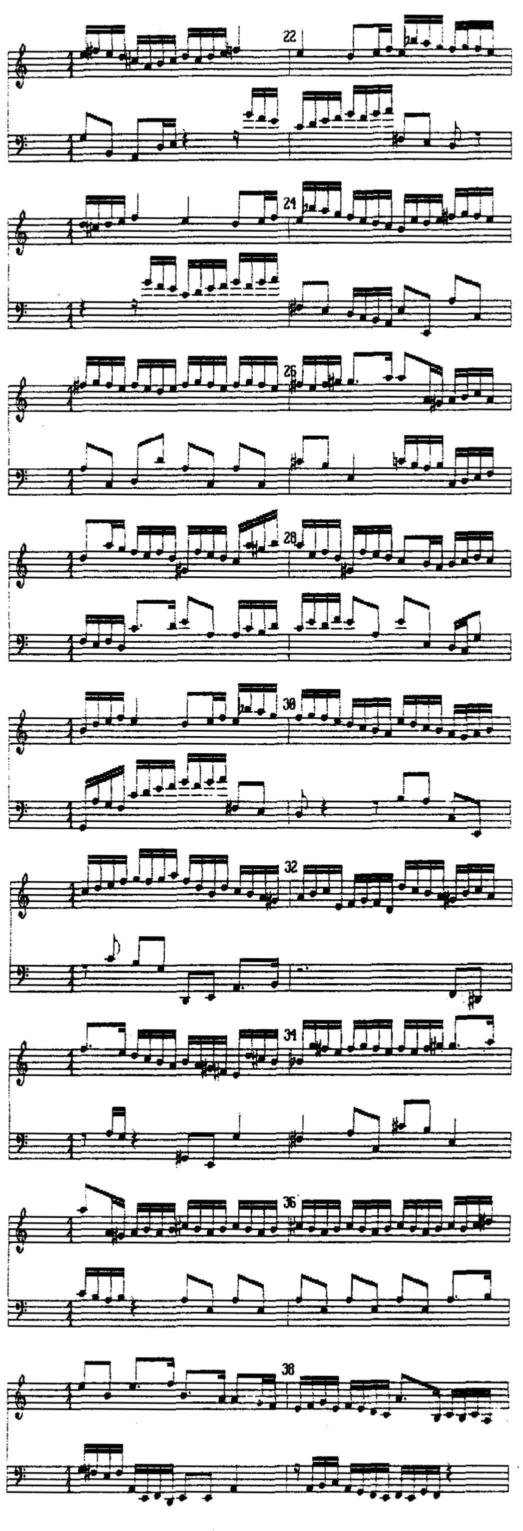
The author is indebted to Mr. Pauli Laine and Mr. Kalev Tiits for the collection of musical source material and synthesis of numerous productions by this method. Thanks are also due to Mr. Kari Torkkola for developing new versions of the program codes.

Caution. This report only describes some of the principal ideas, and *what* was done. Straightforward implementation of these examples may not yet guarantee musically good results. In order to make a practical algorithm, one also has to introduce a great many other details, such as special randomizing processes, diatonic transformations, various checks, etc., not yet discussed above.

References

1. K. Ebcioğlu, "An Expert System for Harmonizing Four-part Chorales", Computer Music Journal, vol. 12, no. 3, 43-51 (1988).
2. T. Kohonen, "Dynamically Expanding Context, with Application to the Correction of Symbol Strings in the Recognition of Continuous Speech", Proc. of the Eighth International Conference on Pattern Recognition, Paris, France, October 28-31, 1986, pp. 1148-1151.
3. T. Kohonen, "Self-Learning Inference Rules by Dynamically Expanding Context", Proc. of the IEEE First Annual International Conference on Neural Networks, San Diego, Cal., June 21-24, 1987, pp. II-3 - II-9.





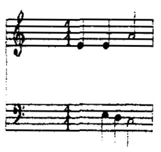


Fig. 3