Pseudo-Genetic Algorithmic Composition

Harald Schmidl
Department of Computer Information Technology
Central Carolina Community College
Sanford, North Carolina, USA

Abstract ***This paper presents a realtime music composition system based on a theme and variation approach. The system uses MIDI and allows up to sixteen instruments to play together. Each instrument requires input of short MIDI files containing musical sequences. Operations inspired by genetic algorithms generate new sequences. Opposed to true, bit-level genetic algorithms the operations modify concrete note properties. This approach facilitates the generation of new sequences within the context intended by the user. Based on the input files the system establishes a list of filters for the purpose of selection. Selection becomes a straight-forward task that affords no user intervention. Scheduling inserts an entire selected sequence into an output queue at a time.***

Keywords: Algorithmic music composition, genetic algorithms, realtime execution, interactive music performance.

1 Introduction

Algorithmic composition uses strategies that emulate aspects of the traditional, human music composition process. A composer uses a set of compositional rules, experience, style, and sometimes intuition to determine the instruments, melodies, harmonies, and rhythms in a piece. An often cited early example of algorithmic composition is Mozart's "Musikalisches Würfelspiel" [1]. A computer is therefore not strictly necessary.

There are great examples of music composed by humans without computers. Why is there an interest in algorithmic composition despite of that? Besides the personal enjoyment of engaging in it there are tangible advantages and applications [2, 3, 4]. Some composers use it as a means of facilitating, enhancing, augmenting, or speeding up the creative process [5]. One can send a computer program on a search for appealing musical parts and complete or refine them manually. Taking over tedious tasks of searching and trying combinations are exactly some of the applications where computers are helpful. A clearly designed algorithm yields results that are easily reproducible. Furthermore, the computer does not expect royalties. Entertainment-oriented applications could benefit from the ability to compose their own versus using commercial music.

In the following text we will elaborate on some relevant, previous work in section 2. Section 3 explains our approach in detail. Section 4 presents a brief run-time analysis. We conclude this paper in section 5.

2 Background

Existing approaches to algorithmic composition use rule-based AI [6], neural nets [7], Markov chains [8], grammars [8], and genetic algorithms (GAs) [9, 10, 11, 12, 13, 14, 15]. Since our work is based on them we present a brief summary of GAs [16].

2.1 Genetic Algorithms in Music Composition

When applied to music, GAs represent the musical material as strings of bits. The GA starts out with an initial parent pool of material. To derive new generations the genetic operations used most commonly are mutation and crossover. Mutation randomly flips bits in a string. Crossover copies parts of two strings together to form a new one. The fittest samples in a generation get priority for reproduction.

Use of GAs for music composition has the advantage of naturally inducing the evolution of new material. One can create material drastically different from the initial parents. On the other hand, if desired, one can only accept material that retains some resemblance to the initial parents. The latter is reminiscent of a theme and variation approach

as used in different, traditional composition styles, such as canon, fugue, or counterpoint.

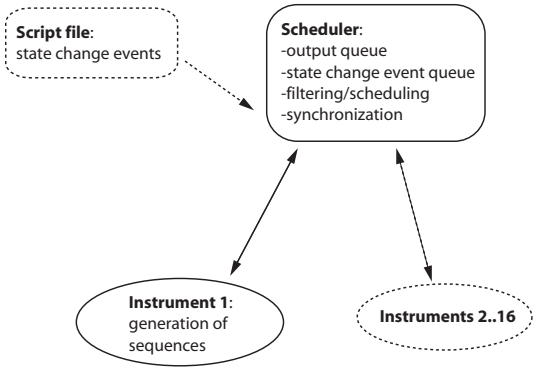

Figure 1: Main system schematic. Optional components are dashed.

Evolution seems to work reasonably well in nature. However, copying nature verbatim will not automatically give an ideal solution. If birds have feathers then why don't we put feathers on airplanes also? In addition, observing the world around us shows that mutation does not always lead to desirable results. When using GAs, selecting good samples is not a trivial task. This applies especially to music, which is a highly subjective domain and often a matter of taste. Selection in GA-driven music composition frequently involves user input. *GenJam* [9] requires keystrokes to indicate the fitness of a currently playing sample. *Variations* [10, 11] has the user enter valid note transitions and intervals manually.

Opposed to existing works we introduce an augmented version of GAs. We devise operations that ensure the generated material lies within the user's intended context. We focus on minimizing human input, avoiding randomness, and ensuring realtime execution. A framework for driving the output interactively is in place. We will now explain the system's components in detail.

3 Our Approach

To get a better picture of our work the reader may access a website with output samples in MP3 format[1]. Our system's central modules are the scheduler and the individual instruments. Figure 1 gives an overview. The instruments generate and maintain all musical material as sequences of MIDI events. MIDI is a protocol for electronic music devices [17]. It provides sixteen channels. Each channel usually represents a different instrument. A note played on a channel consists of a note-on and a note-off event. The events are simple data structures which hold the channel number, note value (pitch), note velocity (volume), and a time stamp. The difference between time stamps of a note's note-on and note-off events determines the note's duration. MIDI is a keyboard-oriented protocol. It performs superior for this class of instruments. Other instruments may lack control of timbre [18].

All instruments create their own generations of sequences. Generations start out with a set of initial parent sequences. To create new sequences we employ the following operations which modify pitch, velocity, and time stamps in a parent sequence:

- replicate
- reverse
- rest
- crossover
- synchronize
- three versions of shuffle
- three versions of mutate

The user can enter any combination of at least one and up to sixteen general MIDI instruments. At least one parent sequence must be input per instrument as a MIDI file, but multiple inputs are possible. The user can also choose which operations an instrument should use to create new sequences.

The scheduler takes care of all global control. It governs the tempo and key the instruments play in. It also selects sequences from each instrument for insertion into an output queue. It processes events in the output queue, synchronizes them by utilizing the computer's internal clock, and sends them to a connected MIDI output device when they become due.

An optional script file serves for scheduling state change events. Currently possible state change events are tempo and key changes, activation or deactivation of an instrument, and end of song. State change events can occur at distinct measures or beats within a song. We place state change events in a separate queue which the scheduler processes synchronously to the output queue.

[1] http://www.cs.miami.edu/~harald/pgac

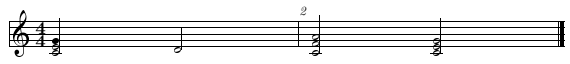

Figure 2: A simple example score.

Figure 3: Patterns after mapping the example score. We indicate pitches by numbers and note durations by line lengths.

3.1 Filters

Jacob describes a way of using filters for selection [10, 11]. His work requires the filters to be input by the user before the GA starts running. To select fit samples he applies the filters at a fixed sampling rate. This approach is prone to missing fine detail if the sampling rate is not small enough. We follow a similar idea but do not require filters to be user input, nor do we utilize a fixed sampling step size.

Once initialized our system analyzes all parent sequences and establishes a list of filters automatically. These filters encode notes that would sound together if the instruments played their user-entered parent sequences simultaneously. The thought is that the initial parent sequences should match according to user intentions. The latter can range from "pleasant" to atonal.

To find the filters we first map all pitches in the parent sequences to a scale of twelve semitones, i.e. an octave. We represent C by 0, $C\sharp$ by 1, D by 2, and so on. We then overlay the resulting sequences and find patterns of notes sounding together at any time. Each distinct pattern becomes a filter. We encode a filter as a string of twelve bits. Each bit indicates whether a note is present or not with a value of 1 or 0 respectively. A filter starts out with all bits set to zero. If a C is included in a pattern we set the least significant bit in the corresponding filter to 1. If a $C\sharp$ is included we set the next bit to 1, etc. Note that in the actual implementation a filter is a 16 bit integer with 4 bits unused.

To illustrate the building of filters assume that the score depicted in figure 2 is the result of overlaying the initial parent sequences of some instruments. Figure 3 shows the patterns yielded after mapping the notes in the example score. We sweep through the patterns from left to right. The analysis will generate two filters: 000010010001 and 001000100001, representing the patterns (0, 4, 7) and (0, 5, 9) respectively.

Only at least two simultaneously playing notes create a filter. Hence, the lonely D with associated note value 2 does not appear in any filter. Note also that there are no separate filters for the patterns (0, 4) and (0, 4, 7). The latter contains the former. Finally, we remove all duplicate filters. Although there are two $Cmaj$ chords in the score, the very first and last chords, the pattern (0, 4, 7) inserts only one filter.

3.2 Filling a Generation

After establishing filters each instrument immediately creates its first generation of sequences. To do so we apply a subset of the earlier stated operations to the instrument's parent sequences. A generation is a list of sequences with a zero-based index.

3.3 Hit-Ratio and Scheduling

Next, scheduling of notes for each instrument begins. The goal is to find and schedule the sequence in an instrument's generation which fits best into all other already scheduled notes. To test a sequence we follow an equivalent strategy as outlined for creating the filters in subsection 3.1. We find all patterns that would sound if the sequence was inserted into the output queue and establish reference strings consisting of twelve bits. We compare these references against the filters by a bit-wise "and" operation. If a filter contains a reference we call it a hit:

if filter **and** reference **equals** reference **then**
  declare hit
end if

We measure the quality of a sequence by its ratio r of hits over its number of references:

$$r = hits/references. \quad (1)$$

The user can enter a threshold t per instrument for the minimum hit-ratio required to consider a sequence a good fit. We maintain a variable i per instrument that indicates the next sequence to be tested. Initially $i = 0$ in order to ensure testing starts with the generation's first sequence. After testing a sequence we increment i, $i = i + 1$. We immediately schedule a sequence that achieves $r \geq t$. Note that before a sequence can be inserted into the output queue it must have its time stamps and

pitch values adjusted to the current tempo and key in which the system is playing.

As long as an instrument has notes scheduled in the output queue we suspend all testing and scheduling for it. When an instrument's last scheduled note is processed and sent to the output device, testing immediately resumes. Testing continues until we identify the next sequence that qualifies for scheduling.

When i reaches the last sequence in an instrument's generation without any sequence exceeding t a special case arises, and we schedule the sequence with the overall maximum r. In addition we create a new generation of sequences and reset i to point to the first sequence in it, $i = 0$.

3.4 New Generations, Fitness and Selection

Generations after the first generation start out by having their parents selected through elite and tournament selection. To do so we first evaluate the fitness of all sequences in the current generation. Elite selection chooses sequences with the highest fitness. Tournament selection randomly picks two sequences and chooses the one with the higher fitness.

Elite selection ensures that strong features are passed on. Tournament selection on the other hand prevents that only the fittest sequences survive. Selecting always only the fittest sequences as parents will cause convergence to self-similar features. Once the parent sequences are established we fill the generation according to subsection 3.2.

To calculate a sequence's fitness we maintain three values per sequence: the most recently calculated hit-ratio r, a counter p that indicates how many times this sequence has been played, and a counter s that indicates how many times it has been selected as a parent. We assess a sequence's fitness as a weighted sum of these three values:

$$fitness = w_r \cdot r + w_p \cdot p + w_s \cdot s. \qquad (2)$$

We use default weights with values $w_r = +1$ and -1 for the other two. This will give preference to sequences that fit well with the user's intentions and it will penalize sequences that have been selected and played in the past. Thus, we prevent generations from becoming self-similar. If desired the user can customize the weights.

3.5 Operations

What remains to be explained are the operations we use to fill a generation once its parents are determined.

Replicate operation: The replicate operation creates an exact copy of a parent.

Reverse operation: The reverse operation reverses all notes in a parent.

Rest operation: The rest operation inserts rests of varying durations into a parent. Insertion of rests keeps the output from sounding monotonous, repetitive, and mechanical. Rests break up the flow of notes and make the output less predictable.

Crossover operation: The crossover operation joins subsets of notes from two different parents to form a new sequence. Crossover helps with avoiding repetitiveness and prevents convergence to self-similar sequences.

Synchronize operation: The synchronize operation synchronizes the time stamps of two parents up to the length of the shorter one.

Shuffle operations: The three shuffle operations swap a number of pairs of pitches, velocities, or time stamps of notes in a parent.

Mutation operations: The three mutation operations can change the pitch, velocity, or time stamps of notes. Pitch can only change up or down by exactly an octave. Time stamps can either be halved or doubled. Velocity can assume one of three values corresponding roughly to forte, mezzo forte, and piano.

When we apply any of our operations to parent sequences, a new, modified sequence is returned and appended to the generation. Except for crossover and synchronize, which act on two, all operations act on exactly one sequence.

Each instrument can use its own subset of operations and each operation can be applied multiple times. As a result the generation size is not fixed. It increases with the numbers of parent sequences, operations chosen, and applications made. The choice of a subset of operations will influence the output when paying attention to rhythmic and melodic detail locally. However, the overall, global outcome changes less dramatically. One can generate interesting output with only the shuffle, rest, and crossover operations as long as one applies them consistently.

The operations are based on and motivated by genetics but are not rigorous, traditional GA operators. Our findings show that GAs work well in the sense that they provide a natural approach to evolution of material. True GAs, however, exhibit a

strong trait of randomness through mutation at bit-level. Changes from a parent to a derived sequence can be severe, the generated material may not follow the user's intentions consistently, and selection work increases.

Our operations also allow creation of very different, new material. At the same time they ensure the results stay within the tonal vocabulary of the initial parent sequences. The longer the system runs the more diverse the output becomes. If one desires the repetition of a main theme one can occasionally reintroduce an initial parent sequence as parent for a new generation.

We summarize our system's main steps in the following algorithm:

```
for all instruments do
   enter initial parent sequences
   fill first generation
end for
analyze initial parent sequences and build filter list
while not end of song do
   process state change events
   process output events
   for all active instruments do
      if no more notes scheduled then
         find best fitting sequence and schedule it
         if end of generation reached then
            select parents for next generation
            fill next generation
         end if
      end if
   end for
end while
```

4 Analysis

We will now present a brief run-time analysis. Our goal was to implement a system that is not only capable of realtime execution but that can be used interactively, e.g. in video games, animation, or virtual reality (VR). The typical frame rate for these applications is 30 frames per second (fps). In the case of VR it is preferably even higher. The question is whether our music composition algorithm will leave enough time for handling graphics and animation.

Refer to figure 4 for the following analysis. Assuming a frame rate of 30 fps we time our system for a composition using three instruments with two or three initial parent sequences each. The total duration of the composed piece is approximately 2 minutes and 10 seconds or 3916 animation frames. Figure 4 shows how much of the frame time is consumed by the composition process. We observe that in 3898 frames (99.54%) less than 10% of the frame time goes into music composition. Furthermore, we find that only 10 frames (0.15%) exhibit a cost that takes up more than 30% of the frame time.

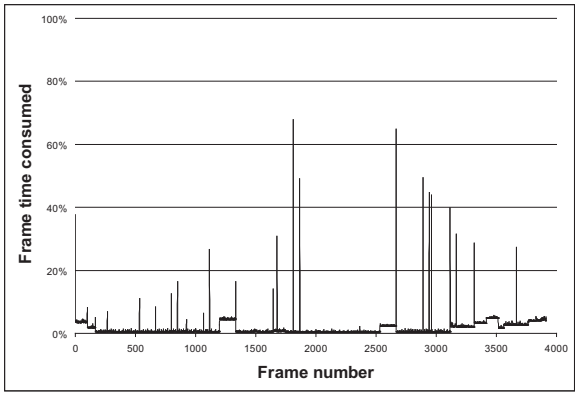

Figure 4: Percentage of animation frame time used by the composition algorithm assuming 30 fps.

Further investigation reveals a surprise. The very first frame is one of the 10 where more than 30% of processing power went into music composition. That is to be expected because during the first frame the system has to initialize itself, create first generations for all instruments, and schedule the first sequences. However, of the remaining 9 frames only 2 have tasks related to scheduling. During the other 7 the scheduler is idle. We interpret these 7 spikes as related to background tasks done by the operating system and not our algorithm. That means only 3 frames (0.08%) have a cost related to music composition that consumes more than 30% of the frame time.

Considering that we did our experiments on an old 866MHz laptop these results give an extremely promising outlook towards adding interactivity. Modern, faster machines and existing real-time animation algorithms should have no difficulties to interface with our system.

5 Conclusion

This paper presented an augmented version of GAs for algorithmic music composition. Encoding all musical sequences through MIDI and applying operations that directly change the MIDI events' pitch, velocity, and time stamps has several advantages. It facilitates creation of new material that follows

the user's intentions. It also avoids the otherwise necessary extra step of translating a bit-string representation into the language of the output device. MIDI, however, has the drawback of lacking control of expression.

A key requirement of our system is to create the initial parent sequences to sound "well" together. This assumption is the basis for the effectiveness of our filtering stage. One must also ensure that the input sequences cover a large tonal spectrum since only notes present in them and their transpositions by octaves will occur in the output. It is furthermore advisable to use several shorter input sequences versus few longer ones.

The strategies for filtering and fitness evaluation produce appealing output and achieve realtime execution. The latter is important in order to ensure the system's capability of interactive performance. Future work will include driving the scheduling and creation of new sequences through interfacing with animation or VR.

References

[1] Curtis Roads. *The Computer Music Tutorial*. MIT Press, Cambridge, MA, 1996.

[2] A. Gartland-Jones and P. Copley. What aspects of musical creativity are sympathetic to evolutionary modeling. *Contemporary Music Review Special Issue: Evolutionary Models of Music*, 22(3):43–55, 2003.

[3] Peter Copley and Andrew Gartland-Jones. Musical form and algorithmic solutions. *Proceedings of the 2005 Conference on Creativity and Cognition*, pages 226–231, 2005.

[4] A. Gartland-Jones. Can a genetic algorithm think like a composer? *5th International Conference on Generative Art*, 2002.

[5] Anthony R. Burton and Tanya R. Vladimirova. Generation of musical sequences with genetic techniques. *Computer Music Journal*, 23(4):59–73, 1999.

[6] D. Cope. An expert system for computer assisted composition. *Computer Music Journal*, 11(4):30–46, 1987.

[7] C. Chen and R. Miikkulainen. Creating melodies with evolving recurrent neural networks. *Proceedings of the 2001 International Joint Conference on Neural Networks*, pages 2241–2246, 2001.

[8] M. Supper. A few remarks on algorithmic composition. *Computer Music Journal*, 25(1):48–53, 2001.

[9] J. Biles. GenJam: A genetic algorithm for generating jazz solos. *Proceedings of the 1994 International Computer Music Conference*, pages 131–137, 1994.

[10] B. Jacob. Composing with genetic algorithms. *Proceedings of the 1994 International Computer Music Conference*, pages 452–455, 1995.

[11] B. Jacob. Algorithmic composition as a model of creativity. *Organised Sound*, 1(3):157–165, 1996.

[12] A. Moroni, J. Manzolli, and F. Von Zuben. Composing with interactive genetic algorithms. *Brazilian Symposium of Computer Music*, 1999.

[13] A. Moroni, J. Manzolli, F. Von Zuben, and R. Gudwin. Evolutionary computation applied to algorithmic composition. *Proceedings of the Congress on Evolutionary Computation*, pages 807–811, 1999.

[14] A. Ayesh and A.Hugill. Genetic approaches for evolving form in musical composition. *Proceedings of the 23rd IASTED International Multi-Conference*, pages 318–321, 2005.

[15] Y. Khalifa and M. Basel Al-Mourad. Autonomous evolutionary music composer. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1873–1874, 2006.

[16] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, MA, 1998.

[17] MIDI Manufacturers Association. *Complete MIDI 1.0 Detailed Specification v96.1*. http://www.midi.org, 2nd edition, 2001.

[18] G. Loy. Musicians make a standard: The MIDI phenomenon. *Computer Music Journal*, 9(4):8–26, 1985.