An Executable Graph Representation for Evolutionary Generative Music

James McDermott
CSAIL, MIT, USA
jamesmichaelmcdermott@gmail.com

Una-May O'Reilly
CSAIL, MIT, USA
unamay@csail.mit.edu

ABSTRACT

We focus on a representation for evolutionary music based on executable graphs in which nodes execute arithmetic functions. Input nodes supply time variables and abstract control variables, and multiple output nodes are mapped to MIDI data. The motivation is that multiple outputs from a single graph should tend to behave in related ways, a key characteristic of good music. While the graph itself determines the short-term behaviour of the music, the control variables can be used to specify large-scale musical structure. This separation of music into form and content enables novel compositional techniques well-suited to writing for games and film, as well as for standalone pieces. A mapping from integer-array genotypes to executable graph phenotypes means that evolution, both interactive and non-interactive, can be applied. Experiments with and without human listeners support several specific claims concerning the system's benefits.

Digital Entertainment Technologies and Arts Track.

Categories and Subject Descriptors

H.5.5 [**Information interfaces and presentation**]: Sound and Music Computing—*Methodologies and techniques, Systems*; I.2.2 [**Artificial Intelligence**]: Automatic programming—*Program synthesis*

General Terms

Algorithms, Experimentation

Keywords

Generative Music, Graphs, Aesthetic Fitness.

1. INTRODUCTION

Attempting to define what does and does not constitute music is often fruitless. Cariani's broad characterisation undercuts the debate: "music entails the temporal patterning of sound for pleasure" [2]. The sheer variety of different types of sound patterns which are deemed pleasurable by different people is one of the most amazing aspects of music. However, Cariani reminds us that in a broad category of music (as in many types of abstract art), pattern is primary.

This category includes those composers, such as Philip Glass and Eric Satie, who are not greatly concerned with improvisation, long melodic lines, and expressive and emotional performance. Instead it is abstract pattern, whether simple or complex, which is of interest. It is this genre of music, broadly construed, which is the long-term goal in this project. Algorithmic techniques such as evolutionary computation tend to fit well into the genre.

Many authors have used evolutionary techniques to generate music of diverse types, and to perform various sub-tasks of musical composition, but none claim it to be a solved problem. There are still many open challenges and opportunities for new evolutionary music research.

We believe that representations are crucial to successful evolutionary computation for music. With this in mind we present a representation for evolutionary music, based on the *NEAT Drummer* project, [7, 8] but extended and tailored to our goals. It is summarised in Fig. 1. An integer-array genotype (a) is mapped to an *executable graph* (b). It maps multiple time-series of numerical inputs (c–e) to multiple numerical outputs interpreted as musical voices (f).

In a sense, the executable graph represents the local or short-term *content* of the music. By varying certain of the input variables in a structured way, longer-term *form* is imposed. Since the input variables are abstract and have no direct or fixed musical interpretation, they can be pre-specified. This does not require any musical knowledge. The structure might also be specified in other ways, for example responding in real-time to in-game events in an adventure game, or to scene changes in a movie.

The multiple output voices are related, not in the sense that one depends on the others, but in that they are all influenced by the same underlying variables and computations. If they are perceived as being *musically* related, this will enforce the essential sense that the voices are aware of each other, and reduce the sense of randomness which is a common fault in generative and evolutionary music. Both interactive and non-interactive modes are available, and they can be combined to good effect.

The remainder of this paper is laid out as follows. Previous work is reviewed in Section 2. The executable graph and other aspects of the representation are described in Section 3. Fitness functions and the interactive mode are described in Sects. 3.3 and 3.4. The system is evaluated in Section 4: each sub-section describes a feature of the system, and as far as is possible provides support through experimental results or references to specific pieces. Conclusions and questions for future work are in Section 5.

2. RELATED WORK

Evolutionary approaches to generative music are well known [1, 11]. Despite some successful and innovative work, the evolution of good music is very far from being a solved problem. A key question, as in all aesthetic domains, is that of the *fitness function*. Four points of view may be distinguished:

1. Non-interactive calculation of aesthetic fitness may be regarded as impossible. It is then natural to allow the user to evaluate fitness, or more commonly to perform *direct aesthetic selection* [12].

2. *Computational features* may be defined and optimised. Sometimes the appropriate target values for such features are derived from analysis of a musical corpus [14].

3. In some research, computational features are defined with a view to guiding the evolution towards plausible areas of the search space, but there is *no true optimisation*. Pieces evolved for thousands of generations may not be expected to be better than those evolved for a few. The features may be regarded as limited surrogates for the composer's aesthetics [4].

4. A very different approach dispenses with the idea of a genome representing a piece of music, and instead deems the artwork to consist of *the dynamics of the entire system* (a population of multiple genomes evolving over time). The fitness function serves to drive the system, but optimisation is not the real aim [5].

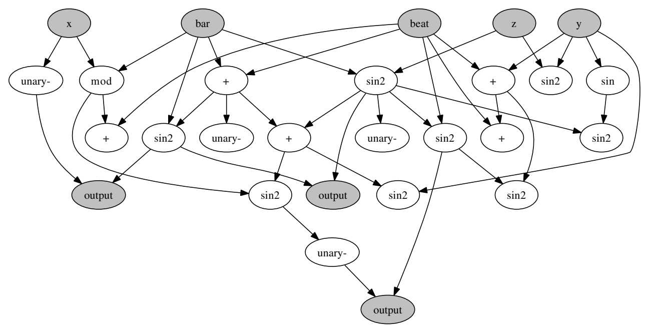
The system to be presented here uses non-interactive fitness as in (3), and interactive selection as in (1).

The *representation* is also crucial. Generative music typically represents music as a function of time. The use of data-flow and signal-flow graphs in generative music has been common for decades, including in software such as Max/MSP (`http://cycling74.com/`) and Jeskola Buzz (`http://buzzmachines.com/`). Two items of previous research are of particular interest. In *NEAT Drummer* [7], networks of functional relationships map input variables, derived from pre-written input music, to plausible, realistic drum tracks. The aim is to automatically provide drums to accompany pre-existing music. The functional networks are created using interactive evolution. They are encoded directly as networks in the *NEAT* (neuro-evolution of augmenting topologies) representation. At each time-step, input variables are derived from the currently-sounding notes of the input music, and mapped to triggers and volume information for a set of drums. In a later version of *NEAT Drummer* [8], inputs are also taken directly from time variables: the current position in the song, in the bar, and in the beat. The user can specify "complex conductors", i.e. time series, as further input variables.

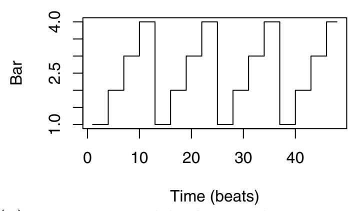
The *Jive* system [12] extends these ideas, again regarding music as a function of time. It uses interactive grammatical evolution to produce arithmetic tree-expressions mapping a

$\boxed{18, 81, 125 : 71, 18, 74 : 66, 101, 113 : \ldots : 17, 66, 46}$

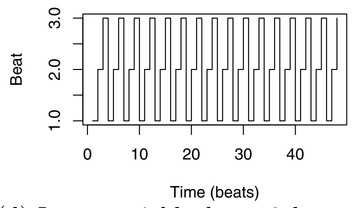
(a) Integer-array genotype is divided into chunks of 3.



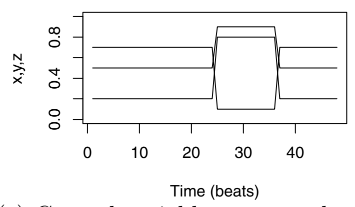
(b) The corresponding executable graph phenotype. Node labels (necessarily too small to read) are function names. Input and output nodes are highlighted.



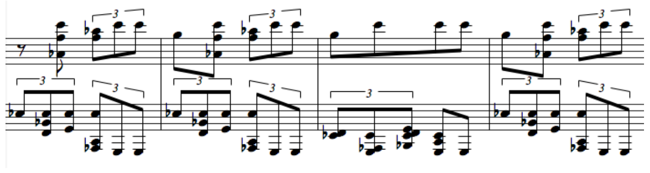
(c) Input variable *bar*. 4 bars per section; 4 sections.



(d) Input variable *beat*. 3 beats per bar.



(e) Control variables x, y, and z impose an AABA structure.



(f) Output in three voices, reflecting the AABA structure.

Figure 1: The representation consists of an integer-array genotype, mapped to an executable graph phenotype, and five time-series of variables which serve as inputs to the graphs. The output is one MIDI voice per output.

Feature	NEAT Drummer	XG
Representation	executable graph	same
Genome	NEAT network	integer array
Node types	Gaussian, sigmoid, etc. (unary)	arithmetic, etc. (various arities)
Edge labels	weights	none
Fitness *	interactive	interactive or non-interactive
Inputs	existing music and conductors	conductors
Conductors:		
primary	time-based	same
complex *	produce motifs	large-scale structure
Output nodes:		
volume*	activity	activity/threshold
pitch	none	second input
Output *	drums	pitched material

Table 1: Similarities and differences between NEAT Drummer and XG, the executable graph system described here. Asterisks mark important differences.

time variable to music. Further numerical variables, controlled by a mouse or 3d controller, are also available. They allow the user to perform and to impose structure, as well as controlling the interactive evolution.

The system we propose builds on and extends both of those mentioned above. It uses directed acyclic graphs (as in *NEAT Drummer*) to represent multiple intertwined functional relationships, and uses both time and abstract control variables (as in both *NEAT Drummer* and *Jive*). The new system also differs from both, in some superficial and some significant ways. It does not use pre-existing music as an input. It uses a different encoding for graphs, somewhat similar to Cartesian GP (CGP) [10]. Node types and graph execution are different. Numerical control variables are used to impose abstract, large-scale structure on the music, rather than being a means of performance, as in *Jive*, or a means of imposing local motifs, as in *NEAT Drummer*. Both interactive and non-interactive fitness modes are available, and can be combined. The similarities and differences *vis-à-vis NEAT Drummer* are summarised in Table 1.

We also draw a distinction between the new system and *Nodal* [9]. There is a similarity between the two, in that both use a graph representation to generate music. However this is largely superficial: in *Nodal* the graph is a finite state machine, which is an entirely different model of computation from the executable graphs used here.

In contrast to systems such as David Cope's *Experiments in Musical Intelligence* [3], ours is a deliberately knowledge-poor representation. It has no knowledge of chords, progressions, melodic or harmonic rules, nor a training corpus. This choice may have both advantages and disadvantages. The motivation is a belief that good music can arise from abstract pattern. An abstract pattern might be expressed through different musical parameters, such as pitch, volume, rhythm, and yet be recognisably the same. With minimal explicit or implicit musical knowledge, the system may have a better chance of surprising its creators. It is certainly simple to implement. On the other hand, a lack of explicit rules allows the system to make some obvious mistakes (for example see Sects. 4.8 and 4.9).

3. THE EXECUTABLE GRAPH REPRESENTATION

In the proposed representation, a piece of music consists of a *directed, acyclic multigraph* together with time-series of values for several continuous *control variables*. In the graph, nodes are labelled by functions such as cos, log and '+'. Edges are directed and unlabelled, multiple distinct edges from one node to another can exist, and cycles are disallowed (taking account of edge direction). Functions have fixed arities, and each node has exactly the number of inbound edges required for the arity of its associated function.

Such a graph can be *executed*. It is first sorted topologically so that later nodes depend only on (i.e. have inbound edges only from) earlier nodes. For each node in the sort order, the function associated with it is then executed using the inputs taken from the outputs of its predecessor nodes. Note that this is a different model of computation from that used in neural networks, where each node is unary, and takes as its input the sum of its weighted inbound edges.

The process begins with five designated input nodes which contain the current values of two input time variables (*bar* and *beat*), and three input control variables. These are similar to "conductors" and "complex conductors" in the *NEAT Drummer* system. Values for the control variables are read from a file. Each node is executed in the order determined by the topological sort. Three designated output nodes contain the overall results of the execution, which are interpreted as MIDI data (see Section 3.2 below for details). The number of control variables and the number of output nodes are easily changed, but are always 3 and 3, respectively, in this paper. By executing a graph many times for appropriate values of the time variables, and corresponding values of control variables, a piece of music is produced.

Each output in an executable graph may be thought of as the root of an arithmetic expression tree, similar to that in genetic programming symbolic regression. Indeed, one can always rewrite an executable graph into multiple disjoint trees, by duplicating nodes and edges as necessary. It is the fact that a graph's outputs share some computations which makes the graph a promising data structure for problems where patterning and re-use are required [10, 7, 8].

3.1 Genotype–phenotype mapping

The executable graphs are encoded as variable-length integer arrays, and a genotype-phenotype mapping is required to produce the graphs themselves. The mapping is similar (but not identical) to CGP [10]. It begins by creating an empty graph, and adding five designated input nodes, each of arity zero. These five nodes will output the values of the *bar*, *beat*, x, y, and z variables.

The genotype is then read from start to finish to add internal nodes. The genotype is divided into chunks of length $m + 1$, where m is the largest arity among node functions. In the current system $m = 2$ (see Section 3.2). Each chunk gives rise to a single new node. The first integer in each chunk gives the node type: it is chosen from a list of functions (listed in Section 3.2, next) including cos, log, and '+' by taking the function with index $g \mod n_n$, where g is the value of the integer and n_n is the number of node functions. Later integers in the chunk are then read to determine the source nodes (which must already exist) from which inbound connections to the new node will be taken, up to the new node's arity. The index of the source node is given by g mod n_i, where n_i is the number of existing nodes. If the arity is less than m, then some integers in the chunk will be redundant: their values then have no effect on the final graph, so they are introns. A later mutation changing the node type might bring them back into play, however.

The process of reading chunks of the genotype and adding new nodes and edges continues until only 3 chunks remain. Each of these final chunks gives rise to an output node. Since the node type is fixed in these cases, the first integer in each chunk is again redundant. As before, the source nodes for each output node's inbound edges are determined by the later integers in the chunk.

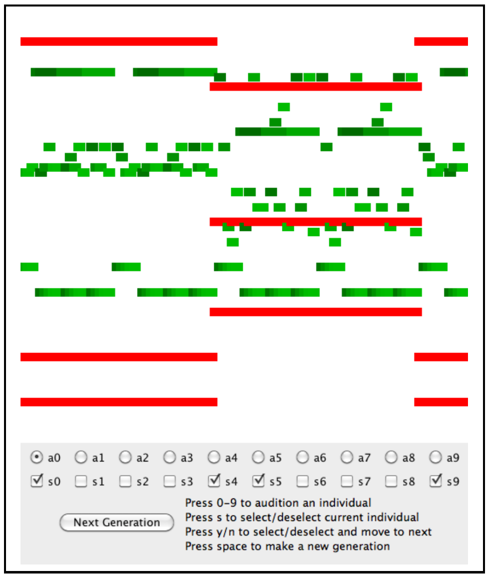
The genetic operators are defined on the genotype. Mutation is per-node and with a given, low probably, alters genes to new random values. One-point crossover is constrained to preserve semantics of crossed-over material. That is, the crossover point in the first individual is chosen randomly, but that in the second is constrained to occur at the corresponding position in a randomly-chosen chunk. That is, $x_0 \bmod m = x_1 \bmod m$, where x_i are the two crossover points. Note that this allows genomes to vary in length, and therefore graphs can grow or shrink in size. As described later (see Section 4.6), this has implications for the complexity of the resulting pieces.

3.2 Node functions

The following node functions are available (arity is given in brackets): sin (1), cos (1), + (2), - (2), * (2), / (2), - (1), mod (2), max (2), sin2 (2), delta (2), edge (1), branch (2). Some of these were chosen as typical of symbolic regression applications of GP. Both / and mod are protected versions: to avoid zero division they return a_0 when $a_1 = 0$, where a_i are the input arguments. Both binary and unary minus are available. Max returns the larger of its two inputs.

Several of the node types were chosen as particulary suited to representing music as a time-based medium. Sin2 returns $a_0 \cdot sin(\text{bar} \cdot a_1)$. Delta returns 1 if its two arguments differ by less than a threshold value, 0.1; otherwise it returns 0. Edge returns 0 if its input is less than zero, but otherwise it returns a rapidly-decreasing function of the input. The input nodes, of arity zero, are bar, beat, x, y, and z. Nodes representing numerical constants also have arity zero: 0.1, 0.2, and 0.5 are available in the current system. Other constants can be created as functions of the existing constants.

Output nodes have arity two. Each output node corresponds to a single musical voice. The two inputs are interpreted as an activity value and a frequency value. Each output node also contains an activity accumulator. When executed, the accumulator decays by a factor of 2, and the new activity value is then added. If the accumulator is now very low, a note-off signal is output, ending any current note on the current output's voice. If the accumulator is higher, but still fails to exceed a fixed threshold value (1), a distinct null value is output, interpreted as "hold"—if a note is already playing, it is allowed to continue, but no new note is added. However if the accumulator exceeds the threshold, then any existing note is ended and a new note-on signal is output. The volume of the new note is determined by the amount by which the activity exceeds the threshold. Its pitch is determined by a sigmoid (tanh) mapping from the frequency value: this is a continuous, monotonic mapping, so small changes in the frequency value lead to small changes



[✓ a0] a1 a2 a3 a4 a5 a6 a7 a8 a9
[✓ s0] s1 s2 s3 [✓ s4] [✓ s5] s6 s7 s8 [✓ s9]
Next Generation

Press 0-9 to audition an individual
Press s to select/deselect current individual
Press y/n to select/deselect and move to next
Press space to make a new generation

Figure 2: The graphical user interface for interactive evolution. Information flows from right to left, representing time. The values of the three control variables (here, long bars) are red on-screen. Output pitches (shorter bars) are green. The top row of buttons chooses which individual to audition, and the bottom row selects individuals. A larger population is filtered to just 10 using non-interactive criteria, reducing the user's workload.

in output pitch, and upward changes in the frequency value lead to upward changes in output pitch. It prevents notes outside a typical range similar to that of a piano. Finally, pitch is mapped to a diatonic scale.

3.3 Fitness functions

Two non-interactive fitness functions are provided.

Feature-Vector A suite of 24 musical features are calculated over each voice of a piece. They include features such as note density, rhythmic variety, and pitch direction. Details are to be found in [14]. A fitness value is calculated by calculating the mean error against target values for some or all of the features. In this paper, the target values were all set to intermediate values, always 0.5 in a [0, 1] scale.

Variety A simple measurement of the variety of a graph's behaviour can be calculated by executing the graph and saving its output while varying its inputs. The inputs' values are drawn from all points in a coarse grid (4 points per dimension). Although the suite of feature vectors contains measures of pitch and rhythmic variety, this method is distinct because it does not depend on a particular file of control variable data.

3.4 Interactive use

The system can also be used interactively. Fig. 2 shows the graphical user interface (GUI). The method is typical of interactive evolution of music, in that just one individual can be auditioned at a time. Selection is binary (yes-or-no).

That is, the user is required only to select favoured individuals, not to provide numerical fitness. This reduced task helps to avoid some user fatigue and is common in previous work [13]. Keyboard shortcuts are available for auditioning, selection, and iteration of the algorithm. The "y" and "n" keys (for "yes" and "no") select/deselect the current individual and begin auditioning of the next. The shortcuts are emphasised in the GUI. The aim is to make the auditioning, selection, and iteration process as streamlined as possible.

A file containing time-series of time and control variables is loaded at startup (future implementations will allow the time-series to be edited, saved, and manipulated in real-time). The time-series is looped continuously. It does not restart when a new individual is auditioned. Since evaluation of complete pieces is time-consuming, a useful strategy is to evolve using a compressed version of the time-series. For example, one might use just an AB structure instead of AAABBA[1]. The user is free to select or deselect at any time and move on to audition the next individual. Bad individuals can therefore often be rejected before they waste too much of the user's time.

Note that the population may be much larger than the 10 shown by default in the GUI. Non-interactive fitness can be used to filter out bad individuals, reducing user fatigue. For example, individuals of low *Variety* (see Section 3.3) can be automatically filtered out. Again, the aim is to avoid user fatigue and frustration, two of the key issues in interactive evolution [11].

4. FEATURES OF THE REPRESENTATION

The representation consists of the variable-length integer-array encoding, the method of "growing" executable graphs, the distinct node functions, and the control variables. It has several potential advantages and disadvantages. In the following sub-sections, the features are described and evidence for each is provided.

We report several types of evidence. In one set of experiments, subjects were asked to listen to specific pieces produced by the system and answer a questionnaire concerning their preference between pairs of pieces, the points in time at which pieces altered most noticeably, and the pairs of tracks which fit together best. 10 subjects were recruited, of varying ages and musical training (from novice to expert). In order to increase the number of pieces exposed to subjects, two complete sets of pieces were evolved and rendered for these experiments. These two sets are available for download and include the questionnaire: `http://www.skynet.ie/~jmmcd/xg.html`.

Control variables followed pre-determined structures such as AAB, ABAB, and so on. Time signatures included 3/4, 4/4, 6/8, 8/8, 9/8, and 12/8. Pieces were roughly 15 to 30 seconds in length. The available diatonic notes were those of F minor, though there was no bias towards playing in this key as opposed to its enharmonic G# major or other equivalent modes.

In other cases, fitness values during evolution provide evidence. Finally, some aesthetic questions have seemed difficult or impossible to investigate objectively, and so the only available approach is to provide references to specific pieces which display desired features or behaviour. It is of course acknowledged that such evidence is purely subjective. These pieces are also available for download, again from the URL given above. Most of these tracks were rendered directly from evolved outputs. In some cases, the tempo or instrumentation was changed. In just one or two, the entire track was repeated multiple times, and one or other voice muted for one of the repeats.

In all experiments reported here, pieces of music were created through purely non-interactive evolution. Individuals were initialised to have lengths between 45 and 90 genes, i.e. between 15 and 30 nodes, plus inputs. The population was 30 and number of generations 20, except where noted (these low numbers were chosen partly because very good music often occurs at the end of such runs, and partly to reflect the restrictions on population size and number of generations in interactive mode). Tournament selection was applied with size 7. Elitism passed-through a proportion of 0.05 of the population. Crossover was one-point, as described above. Mutation re-randomised each gene with probability 0.05.

4.1 Structure is represented in an abstract way

The control variables x, y, and z do not have any direct interpretation such as controlling crescendo or rallentando. Instead, they work together to produce abstract structure. If x varies between one value and another on an alternating schedule of, say, 8 bars, then an ABAB structure will be produced, even though A and B themselves have not yet been specified. If y is varied continuously from a low value to a high value while the value of x alternates, the extra structure will be somehow superimposed on the music even if y has no known interpretation.

The hypothesis is made precise as follows: listeners are able to determine the point in time at which control variables values change. Two sets of four tracks were used, using pre-determined structures such as AABB, ABB, AAABB, and so on, with in every case a single transition from the A sections to the B sections. This transition was imposed purely by a change in the control variables, all changing at the same time. Listeners were asked to identify the most noticeable change in each piece. The question asked was "Listen to track 9. In this track, at what point in time does the music change most noticeably? Use rewind if necessary. Please give a time, in seconds: _ ." The transition point was different for different pieces, since they had different structures and tempos. Therefore listeners did not receive clues about the current piece from previous pieces.

For each such trial, the "error" was calculated as the difference between the point in time at which the control variables changed, and the time given by the listener. The results were quite clear-cut. The mean error was 0.94s, with standard deviation 1.9. The number of errors greater than 1s was just 7 out of 40. This supports the claim that the abstract control variables provide discernible structure.

4.2 Form and content are separated

When holding control variables constant, a simple piece of music will be produced by the increase of the time variables. It will repeat after a short time since they are peri-

[1]This notation for musical structure is standard. Each letter refers to a section of music. A pop song's verse-chorus-verse-chorus structure might be written ABAB. The letters do not correspond to the letter names for notes or chords. Note that although control variables are often arranged into A and B sections throughout the paper, there is no constraint to use just two section types, or to use any clear-cut sections at all.

odic. We can regard this music as the local *content* of the piece, while the *form* or structure is created by varying the control variables. In practice the distinction is not always clear-cut, especially when the control variables move continuously. This reflects the difficulty of separating form and content in (non-generative) music in general. Nevertheless, some interesting compositional techniques become available using this idea. One possibility is to automatically alter the time signature of a piece. This is demonstrated in tracks *Transform 6, 8* in the demo pieces available for download. A simple piece in 6/8 time is automatically extended to 8/8. Note that a transformation in the other direction would be easy to program even in a non-generative representation. This is accomplished by evolving a piece in 6/8 time and then changing the control data to 8/8. The generation or manual editing of such control data is trivial. The result is that the 8/8 piece sounds like a natural transformation of the 6/8 piece. The first 6 beats of each bar are identical between the two pieces, and since the two extra beats are generated by the same graph that generated the previous 6, they seem to fit. The two tracks are clearly related. This type of operation allows a user to think in more abstract terms when composing: large-scale transformations can be tried out trivially, without a great deal of manual editing.

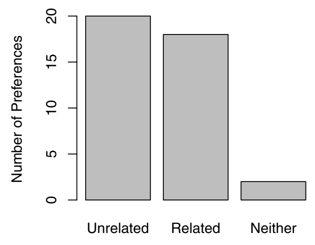
4.3 Multiple voices are related

Since a piece of music is represented by a single graph (together with values for control variables), the calculations performed for the multiple voices share common inputs (time and control variables) and some common nodes and edges. Therefore they may be expected to sound related. For example, when control variables change, all voices react to the change simultaneously. This creates the impression of the music being deliberately written rather than random. Note that the relationship between voices is not that of one voice controlling another: rather, all voices are influenced by their common inputs and calculations. This is the motivation behind the use of graphs, here and in *NEAT Drummer*.

The hypothesis in this experiment is that a piece of music of two simultaneous voices, both derived from the same graph, will sound more coherent (the voices more related) than a "Frankenstein" piece of two voices taken from different graphs. Two sets of four pieces were created. To avoid superficial differences, all pieces shared the same parameters, control data, and so on. Each pair of pieces provided two Frankenstein pieces, by swapping voices, and two normal pieces. A single trial of the experiment consisted of comparing a normal piece with a Frankenstein piece (these labels were not known to the subjects). The question asked was "Listen to tracks 13 and 14. Each is made of two simultaneous melodies. In which track do the two melodies fit together best? Track 13: _ Track 14: _ Neither: _ ."

Each subject performed four such trials. If two evolved pieces p and q consist of voices p_1, p_2 and q_1, q_2, then the first trial for a user compared p_1/p_2 and p_1/q_2, and the second compared q_1/p_2 and q_1/q_2. The third and fourth trials were similar, with entirely new evolved pieces for p and q. In summary, any bias arising from some voices simply being better (as opposed to matching other voices better) was avoided by design.

The results do *not* confirm the hypothesis in this case. In Fig. 3, the original pieces and Frankenstein pieces were approximately equally liked There are at least two possible



(a) Coherence

Figure 3: Pieces composed by the system and "Frankenstein" pieces created by swapping voices from unrelated pieces were approximately equally liked.

explanations for this unexpected result. Firstly, it may have arisen partly because the multiple voices in Frankenstein pieces tended to fit together surprisingly well—as opposed to those in normal pieces fitting together badly. The original pieces whose voices were swapped-over to create the Frankenstein pieces were possibly so close together in the search space that all voices suited each other quite well. Recall that they used precisely the same control data, tempo, and pitch mapping. They were "unrelated" only in being drawn from different graphs. Secondly, several subjects reported difficulty in interpreting this question and understanding their task. It is possible that a different experimental design, in which the normal pieces were evolved to different targets and thus drawn from different areas of the search space, and a better wording of the question, would lead to a different result on this topic. In any case, there is no reason to believe that the executable graph representation produces pieces whose voices are *less* suited to each other than alternative functional representations.

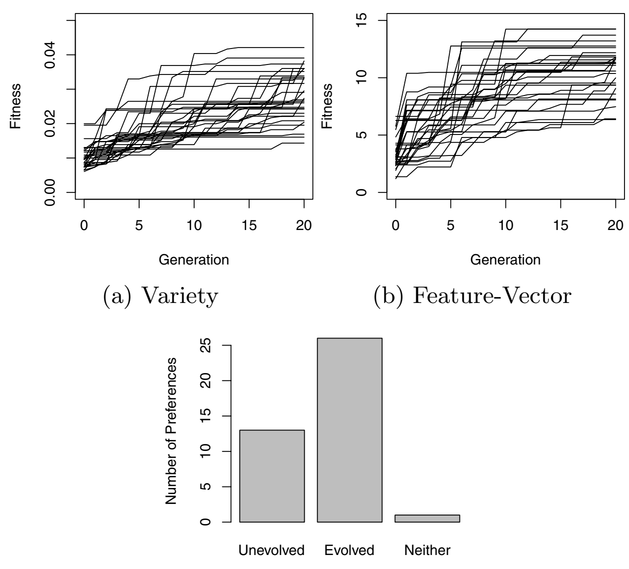
Listening to outputs leaves no doubt that control variables' abstract structure simultaneously affects multiple voices. In general, when one voice changes, they all do. This reinforces the sense of teleology and purpose in the piece, something that is often absent from generative music. The two pieces *Transform 6, 8* demonstrate this clearly. The pieces *Piano 1–3* also appear to have real purpose behind them.

4.4 Natural-sounding syntax and dynamics

The output nodes are quite simple, but seem to be very effective in producing human-like syntax and dynamics. It is the activity/threshold mechanism which is responsible for setting patterns of note-on and note-off commands, and dynamics. Good examples include tracks *Dynamics 1–3* in the demo pieces available for download. There is an obvious improvement over a system such as *Jive* [12], in which dynamics are entirely absent.

4.5 Direct control of duration

Just as musical structure can be controlled by providing time-series of values for the control variables, so can the length of the piece. In itself this point is trivial: almost any generative system can produce pieces of unlimited length which can be cut to give a desired length. However, evolutionary representations are not always controllable in this way. A good example is the *Ossia* system [4], where a geno-



(a) Variety (b) Feature-Vector

(c) Approval

Figure 4: Non-interactive evolution can produce good results even with a small population (30) and few generations (20). In (a) and (b), two non-interactive fitness functions are each improved over the course of 30 runs. In (c), the results of (a) are preferred by subjects over unevolved pieces.

type corresponds to a fixed amount of musical material and can not be easily extended.

4.6 Direct and indirect control of musical complexity

Good music exists in a range between overly patterned and repetitious music, on one side, and randomness on the other. Different styles require different balances between the two. A method of controlling complexity is therefore potentially useful. Controlling the number of nodes is a likely method. Assuming that the *Variety* fitness function is a reasonable surrogate for a complexity measure, our hypothesis is that variety is related to the number of nodes.

Data taken from the same experiments reported in Section 4.7 confirms the hypothesis. Individuals were randomly-initialised to have between 45 and 90 genes, corresponding to between 15 and 30 nodes (plus input nodes which do not require genes). The mean over 30 runs of the best individual's genome size at initialisation was 64, rising to 252 after 20 generations. This increase is caused by selecting for variety, as opposed to being a side-effect of evolutionary dynamics: in runs with effectively random selection, the mean best genome's size increased very slightly, from 65 to 70. Therefore, controlling genotype size via choice of crossover points allows some direct control over musical complexity. The *Variety* fitness function might also be used to provide indirect control, in that one can not immediately produce a given value for variety, as one can for node-count.

4.7 Non-interactive evolution is effective

Figure 4 (a–b) shows best-of-generation fitness over 30 short evolutionary runs using each of the two non-interactive fitness functions *Variety* and *Feature-vector* (explained in Section 3.3). Despite the small population size (30) and number of generations (20), large improvements in fitness can take place during a run. This is perhaps a surprising result. It suggests that even the short runs characteristic of interactive evolution may allow the user to improve the population, or direct it towards desired areas of the search space.

Figure 4 (c) tells the same story from the point of view of user approval of pieces generated by non-interactive evolution. Two sets of pieces were created, both using the same structural inputs. One set was created by random generation, as in the initialisation stage of the evolutionary algorithm, with no selection, crossover, or mutation. The second set was created by evolving a population of size 30 over 20 generations with selection driven by the *Variety* fitness function. The pieces were presented to the same users, in the same sessions as in Sects. 4.1 and 4.3. The question asked was "Listen to tracks 7 and 8. Which do you prefer? Track 7: _ Track 8: _ Neither: _ ." Users' preferences among the two sets of pieces are summarised in the barchart (Fig. 4 (d)): pieces evolved with this fitness function, even in a small population and over few generations, are a clear improvement over unevolved pieces.

4.8 Over-complexity appears as randomness

A piece of music in which the underlying patterns are too complex to be understood or perceived might as well be random. This is one common failure mode of the system. The *Variety* fitness function in particular rewards large numbers of distinct pitches. In typical runs, this leads the population towards areas where variety is high, but not too high. In some runs, variety is more strongly maximised, leading to random-sounding music. A better fitness function might aim for a middle-ground of variety, rather than maximising it. This possibility will be explored in future work.

4.9 Repeated notes are too common

The most common weakness of the system is the production of a long string of short notes of identical pitch. This is a consequence of the activity/threshold mechanism in the output nodes. A constant value for the output node's frequency input is easily achieved, so a string of short notes of the same pitch will occur whenever high activation values arrive. Many randomly-generated graphs fall into this failure mode, and so interactive users typically have to deal with several very weak pieces of this type in early generations. In non-interactive mode, the fitness functions tend to weed out such individuals, though it is not uncommon for them to survive into the final generation. Allowing a larger population size and number of generations would would help to alleviate this problem at the expense of producing more random-sounding pieces, as described in the previous section. Again, highly tuned fitness functions might help to solve this problem, but they are not the focus of this paper.

5. CONCLUSIONS

In the representation for evolutionary music described here, variable-length integer genotypes are decoded to executable graph phenotypes. Numerical values for time and control variables are input to the graph, and the output is interpreted as MIDI data. After relatively little optimisation with simple, non-interactive fitness functions, the resulting music is surprisingly good. Many pieces have been created which (subjectively speaking) exhibit well-formed structure, teleology, and coherence among multiple voices. Sometimes they work well as standalone pieces, sometimes only as the seed of a new piece. They are surprising and novel in that they exhibit qualities the authors do not typically use in their hand-written music. The melodic, rhythmic, harmonic and dynamic aspects of the music often feel surprisingly human. The quality of the results is inevitably subjective, so the reader is encouraged to download and listen to the demo tracks available from `http://www.skynet.ie/~jmmcd/xg.html`. The source code of the system is also available.

The graph representation described in this paper is influenced by *NEAT Drummer* [7, 8], though it also differs in some ways (see Section 2). The fact that good results are achieved despite several dissimilarities, both superficial and significant, reflects the richness of the original ideas.

The system has been evaluated objectively in as far as is possible. Both purely computational experiments, and some with human listeners, have been carried out to test the various claimed benefits of the system, and have in general confirmed them.

After evaluating the system in terms of EC design choices, we now turn to the choice of EC itself. The GA (with variable-length integer genotype) seems to be the right choice for this problem. The non-interactive fitness functions used in this work seem unlikely to be amenable to the calculation of derivatives, as required by gradient-based methods of search and optimisation. A similar problem exists for binary (yes-or-no) selection as used in interactive mode. In this case, the lack of a ranking among the selected individuals makes evolutionary methods such as particle swarm optimisation unsuitable. However, recombination-based evolutionary methods, such as the GA, are applicable even in these circumstances. The abstract processes of selection, recombination, and variation seem to match some models of creativity well [6], and they are intuitively well-understood by non-technical users.

The separation of form and content allows a new approach to a common problem in interactive evolutionary art and music. Users of such systems often find that a good individual requires a simple edit, addition, or deletion in order to be much better, but the system fails to provide the desired change. This is frustrating in that the user user may feel it would be better to abandon evolution and work manually. Usually, manually-edited pieces can not be re-imported to the population. The ability in our new system to cut out or repeat a section of the time-series of control variables allows the user to make a quick fix and continue evolution.

The system's method of specifying music as a function of time and control variables has great potential for creating generative music for film and video games. Non-generative music has a disadvantage in these contexts in that it does not respond to changes in the visual material. A film director making a last-minute cut to a scene might require the music to be re-edited to fit. A game designer might be unhappy with abrupt changes in music to suit sudden events. The system we describe has the potential to address these problems, because the executable graph is responsive to input. Changing length or structure can be accomplished by simple editing of the existing time-series of control variables. A single piece of music can be designed to react to in-game events without losing its original character.

One branch of future work will seek to develop applications such as those mentioned above. Another will improve non-interactive fitness functions. The most common faults in current outputs will be identified and fitness functions written to avoid them. Alternative methods of specifying control variables will also be investigated, including real-time control (as in *Jive*), a specialised time-series editor, and the derivation of control variables from other types of structured time-series.

Acknowledgements

JMcD gratefully acknowledges funding from the Irish Research Council for Science, Engineering and Technology, co-funded by Marie Curie Actions under FP7. Thanks are due to the anonymous reviewers for helping to improve this work.

6. REFERENCES

[1] P. J. Bentley and D. W. Corne, editors. *Creative Evolutionary Systems*. Morgan Kaufmann, 2002.

[2] P. Cariani. Temporal codes, timing nets, and music perception. *Journal of New Music Perception*, 30(2), 2001.

[3] D. Cope. Computer modeling of musical intelligence in EMI. *Computer Music Journal*, 16(2):pp. 69–83, 1992.

[4] P. Dahlstedt. Autonomous evolution of complete piano pieces and performances. In *Proceedings of Music AL Workshop*, 2007.

[5] A. Eldridge and A. Dorin. Filterscape: Energy recycling in a creative ecosystem. In *EvoMUSART*, pages 508–517. Springer, 2009.

[6] D. E. Goldberg. The race, the hurdle, and the sweet spot: Lessons from genetic algorithms for the automation of design innovation and creativity. In P. J. Bentley, editor, *Evolutionary Design by Computers*. Morgan Kaufman, 1999.

[7] A. K. Hoover, M. P. Rosario, and K. O. Stanley. Scaffolding for interactively evolving novel drum tracks for existing songs. In *Proceedings of EvoWorkshops*, volume 4974 of *LNCS*, page 412. Springer, 2008.

[8] A. K. Hoover and K. O. Stanley. Exploiting functional relationships in musical composition. *Connection Science*, 21(2):227–251, 2009.

[9] J. McCormack, P. McIlwain, A. Lane, and A. Dorin. Generative composition with Nodal. In E. Miranda, editor, *Workshop on Music and Artificial Life (part of ECAL 2007)*, Lisbon, Portugal, 2007.

[10] J. F. Miller and P. Thomson. Cartesian genetic programming. In *EuroGP*, pages 121–132. Springer, 2000.

[11] E. R. Miranda and J. A. Biles, editors. *Evolutionary Computer Music*. Springer, 2007.

[12] J. Shao, J. McDermott, M. O'Neill, and A. Brabazon. Jive: A generative, interactive, virtual, evolutionary music system. In *Proceedings of EvoWorkshops*, 2010.

[13] H. Takagi. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proc. of the IEEE*, 89(9):1275–1296, 2001.

[14] M. Towsey, A. Brown, S. Wright, and J. Diederich. Towards melodic extension using genetic algorithms. *Educational Technology & Society*, 4(2), 2001.