Noname manuscript No.
(will be inserted by the editor)

Evolving the process of a virtual composer

Csaba Sulyok · Andrew McPherson · Christopher Harte

Received: date / Accepted: date

Abstract In this paper we present a genetic programming system that evolves the music composition process rather than the musical product. We model the composition process using a Turing-complete virtual register machine, which renders musical pieces. These are evaluated using a series of fitness tests, which judge their statistical similarity against a corpus of Bach keyboard exercises. We explore the space of parameters for the system, looking specifically at population size, single- versus multi-track pieces and virtual machine instruction set design. Results demonstrate that the methodology succeeds in creating pieces of music that converge towards the properties of the chosen corpus. The output pieces exhibit certain musical qualities (repetition and variation) not specifically targeted by our fitness tests, emerging solely based on the statistical similarities.

Keywords evolutionary algorithms · music generation · music evaluation · corpus-based similarity tests

1 Introduction

Music composition is a fundamentally subjective process. It requires not only the creation of new musical ideas but also the critical assessment and refinement of newly created material in order to produce a completed work. Thywissen [35] describes it as 'an aesthetic search through the space of possible structures that satisfy the requirements of that process'. Composers spend years training and perfecting their technique; to create new music successfully, both experience of the composition process and knowledge of existing 'good' music is required. The composer's judgement as to whether a new musical idea is good or bad will be a subjective decision based on their knowledge and memory of previous pieces (see Figure 1). As their creative process evolves, so too should the quality of their compositions.

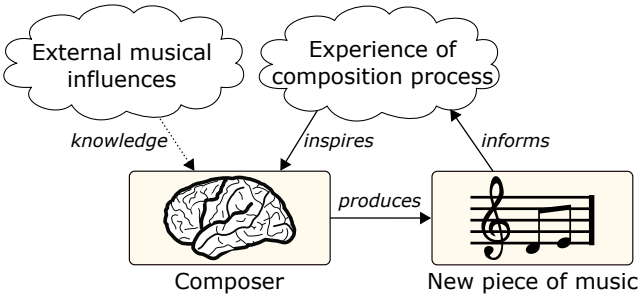


Fig. 1: The creative process of a composer is informed by both experience of music in general (external influence of others' music) and experience gained through practice of the composition process itself.

With the exception of formal tuition, the external knowledge gained from music written by others is generally indirect; a listener experiences only the output of the composition process, without being able to observe the steps taken to create it. Even reading and analysing the scored music for a piece gives little clue about the

Csaba Sulyok
Queen Mary University of London, Mile End Road, London, E1 4NS
E-mail: csaba.sulyok@gmail.com

Andrew McPherson
Queen Mary University of London, Mile End Road, London, E1 4NS
E-mail: a.mcpherson@qmul.ac.uk

Christopher Harte
Melodient Ltd. Leeds, UK
E-mail: chris@melodient.com

composition process itself. When someone tries to compose music for the first time, the structures inferred from these outside influences provide the foundations for new works. After the initial attempts, later compositions may be more directly inspired by the composer's own previous pieces. The composer has full knowledge of the composition process used in their previous works and can refine this process over time.

Viewing the development of a composer's skill as an evolving process intuitively suggests that evolutionary computation techniques should find utility in algorithmic music composition. In this paper we model the composer as a Turing-complete virtual machine (VM); the mind is viewed as a computer with a predefined set of instructions representing real-world atomic steps used to produce a piece of music. Each composition is the result of executing a specific program on the machine and it is these programs that are the subject of our evolutionary process. The output of the VM is a byte array which only receives context once we view it as a musical piece. As a result, the system is not limited specifically to producing music; the evolutionary strategy for its programs could represent any creative process.

In our context, the outputs of the system are interpreted as musical pieces and assessed by judging their statistical similarity to a corpus of real music (in this case Bach's *Inventions and Sinfonias*). The corpus represents the set of our virtual composer's external influences and, as in the real world, we have no direct knowledge of the composition process that produced that music. The statistics we use to judge similarity to the corpus have been chosen based on a number of general musical traits they may represent. While the tests are aimed at musical properties, the comparison process itself is applicable to any corpus-based evolutionary process.

Using evolutionary models for composing music is nothing new; examples date back to the early 1990s (see [16,14,17,24,19] and for an exhaustive review [30]). However, the subjective nature of music quality makes defining appropriate machine fitness tests difficult [16, 24,25,39]. To reduce the otherwise vast solution space of all possible music, many past approaches have focused on evolving one particular musical property, e.g. melody [1,37], harmony [24,28,6], rhythmic patterns [23,9,27] or performance elements [5]. Other approaches to the problem include incorporating artificial neural networks [14,4] and evolving musical grammars [29,20].

An alternative way to guide the search for 'good' music has been to provide favourable initial conditions. Waschka's *GenDash* [39] uses an initial population of musical material defined by the user themselves to help evolve and refine their ideas rather than have the machine compose music without supervision; Waschka has released a large number of pieces created with the aid of this program. Eigenfeldt's *Kinetic Engine* [10,11] evolves rhythmic and melodic patterns; its initial population being derived from a pre-computed offline analysis of a corpus. In contrast, rather than seeding the initial population, Donnelly and Sheppard [8] evolve four-part harmony by providing the first chord to develop from upon initialisation of the algorithm.

Other approaches have used *interactive genetic algorithms* (IGAs) which rely on human feedback for fitness evaluation. Tokui and Iba [36] and Horowitz [18] applied IGAs to evolve rhythm patterns, while Jacob [19] used it for evolving multiple musical properties. One of the early music-related IGAs is Biles' *GenJam* [2]: a program capable of evolving improvisatory passages in realtime. Other examples of IGAs include the MIDI-based *GeNotator* by Thywissen [35] and *SBEAT3* by Unemi [38]. More recently, MacCallum et al. [21] created the online *DarwinTunes* community to crowdsource human feedback for an evolutionary composition system with members of the public choosing which music pieces will be selected for breeding.

The system we present here differs from previous work in the literature by incorporating linear genetic programming [3] elements via a VM, effectively evolving the composition process rather than musical pieces themselves. The way in which we utilise a corpus is also novel in that we use it to assess fitness rather than deriving our initial population from it.

This paper builds on our previous research work in [34] using an evolutionary algorithm for music composition by exploring the space of different system parameters more fully. Our results show that musical pieces with reoccurring patterns and motifs emerge but other musical properties such as harmony are currently lacking, suggesting the need for new fitness tests perhaps more directly inspired by music theory.

The paper is structured as follows: Section 2 provides the overview of the evolutionary system while Sections 3, 4, 5 and 6 provide in-depth details of its building blocks. Section 7 presents the tested configurations; Section 8 analyses and discusses the results and Section 9 draws conclusions and describes possibilities for future exploration.

2 System Overview

In this research, we cast the action of composing a piece of music as a process running on a Turing-complete virtual machine. We define a *genetic string* as the initial

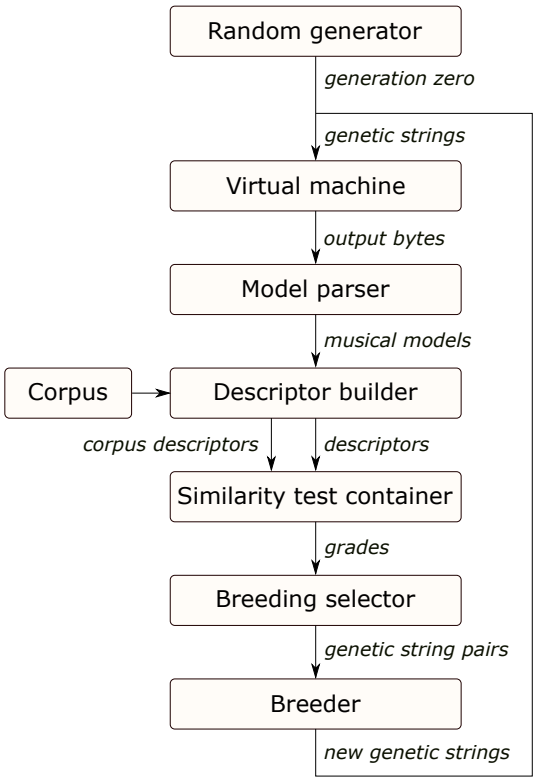


Fig. 2: Overview of the evolutionary algorithm: The population consists of our genotype genetic strings (see Section 2) that represent programs which are interpreted by the VM (see Section 3). The output of the machine is used to build musical models (see Section 5) that are the phenotype. Statistical similarities between these models and a corpus of real music determines their fitness (see Section 6) and therefore their chance of survival and breeding.

condition of the VM comprising values for all registers and memory segments; the outcome being entirely determined by this initial state. The development of the composer's skill then becomes a genetic programming problem: the genotype is the genetic string presented to the VM and the phenotype its musical output. In such a system, a process executing on the VM can hold internal structuring rules and information that are not directly visible in the final musical phenotype. Whorley et al. [40] address this point in the context of melody harmonisation, the exact steps towards which cannot be known just by listening to the resulting piece of music. Decoupling the genetic program and the resulting musical phenotype in this way allows for the emergence of complex structures not possible in more constrained musical models. For example, a conditional branch instruction executed on the VM could cause the repetition of a single note or section, or perhaps even the entire piece depending on where it occurs in the program. While the musical possibilities of this approach are effectively unconstrained, the resulting space of possible outcomes becomes correspondingly vast making it hard to analyse. Careful design of the fitness evaluation strategy is therefore necessary.

Figure 2 shows the outline structure of our evolutionary composition system. The output of the VM is a byte array that is parsed by a *model parser* to create our phenotype, a musical model. The structure of this model is independent from the structure and mechanism of the VM. This two-stage approach to rendering the model contrasts previous musical evolutionary systems in that the structure of the genetic string does not directly represent that of the phenotype. An additional benefit of separating the parsing step from operation of the VM is that system could easily be adapted to different tasks simply by changing the parser and altering the fitness tests accordingly; the rest of the system remaining unchanged.

As mentioned in Section 1, the composer's creative process cannot operate in a vacuum; it is necessarily dependent on the influence of works from previous composers. To achieve this, our evaluation process employs fitness tests that judge the similarity between certain properties of the current musical phenotype and those of a chosen corpus of existing music [1]. These tests focus on the underlying statistical properties of a model rather than the similarity of the data itself; it is therefore possible for a model to achieve high grades without being identical to a member of the corpus as long as it shares certain traits captured by those statistics. To produce these statistics, we subject models to a series of transform methods to produce a *descriptor* (as performed by the descriptor builder block shown in Figure 2). We refer to the complete set of tests as the *similarity test container*.

Based on the assigned grades, the *breeding selector* chooses pairs of candidates for crossover and/or survival and the *breeder* splices these pairs to create offspring. The crossover and mutation process operates on the genetic strings producing a new generation of programs for the VM to execute.

3 Virtual machine

Our VM is an emulation of a Turing-complete von Neumann register machine [26]. This choice of architecture

[1] It should be noted that the corpus information is only used to inform our fitness evaluation. If desired, tests based on alternative criteria could be readily substituted without requiring changes to any other part of the system.

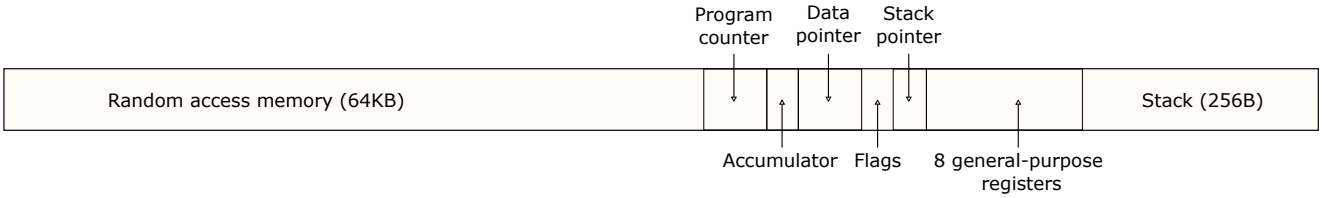


Fig. 3: Structure of the genetic string which can be fed to our VM. It contains a 64kB random access memory (RAM) and a 256B stack, which can be addressed by a 16-bit program counter and 8-bit stack pointer respectively. Other registers include eight general-purpose 8-bit registers, an accumulator, a 16-bit data pointer and a set of flags.

allows maximum flexibility in the system since a process can alter its own code while it is running. The VM contains the following memory segments (also shown in Figure 3):

- *64kB random access memory (RAM)* - contains the actual program. Since this is a von Neumann machine, the RAM is also part of the data used by the program, therefore it may overwrite itself during execution.
- *16-bit program counter* - a register that points to the location in the RAM where the next instruction resides; it is capable of addressing any position in the RAM and is allowed to wrap around during execution.
- *256-byte stack* - a segment of memory where the program may push or pop data from the registers or the RAM.
- *8-bit stack pointer* - a register that points to the location in the stack where the program will push to/pop from.
- *8-bit accumulator register* - a register with which arithmetic/logical instructions are performed.
- *16-bit data pointer* - a general address register capable of addressing any position in the RAM.
- *A set of flags* - a status register which contains additional information on the state of the processor; currently only a carry flag is used in arithmetic operations.
- *Eight 8-bit general purpose registers.*

The fetch cycle of the VM is illustrated in Figure 4 showing an example of reading a byte from the RAM, locating the mapped instruction, incrementing the program counter and running the instruction. When an output instruction is encountered, bytes stored in a register or at a location in memory are output from the VM by appending them to a dynamically-sized output array. Execution is terminated either by reaching a halt instruction or after the machine has processed a preset maximum number of instructions or output bytes; the latter conditions being added to account for infinite loops. The VM process is completed in full before the output array is interpreted by the model parser.

The instruction set contains the following types of instructions:

1. *Data transfer* - copy a segment of the RAM or a register to another segment of the RAM or to a register;
2. *Arithmetic & logic* - perform simple arithmetic and logical functions on the accumulator register and, optionally, another value from the RAM or a register;
3. *Branching & conditional instructions* - alter the program counter to point to a different location, optionally based on a condition;
4. *Machine control* - internal instructions such as halting or pushing/popping using the stack;
5. *Output instructions* - send values from the RAM or a register to the output byte array.

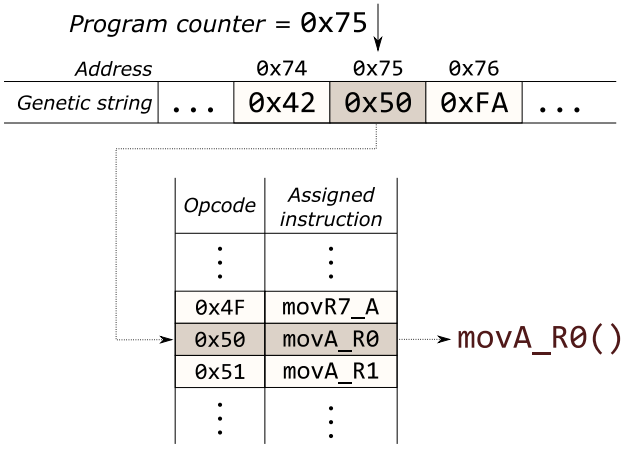


Fig. 4: Interpreting an instruction in the VM: an 8-bit value is fetched from the RAM at the address pointed to by the program counter, the mapped instruction is identified, the program counter is incremented and the instruction executed.

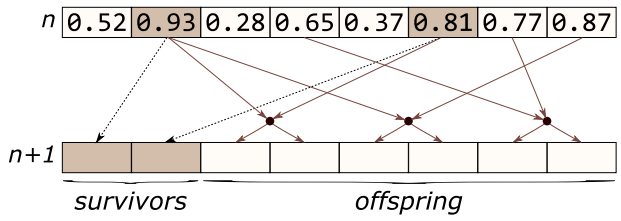


Fig. 5: An example of choosing candidates to build generation $n+1$ from generation n. The numbers in generation n represent each individual's overall fitness. The darker individuals have been randomly selected for survival from a distribution based on their grades and age, while the remainder of the population is filled by choosing pairs of parents for crossover.

A random genetic string contains all the different opcodes in a uniform distribution. However, interpreting such an input does not guarantee their uniform occurrences during execution since branching instructions may form loops and data transfer instructions may overwrite sections of the RAM. The algorithm may therefore evolve to favour certain instruction types more than others, given they lead to a more successful phenotype; a detailed analysis of instruction type occurrence in our tests can be found in Section 8.4.

In our current experiments, we have defined two different instruction sets, differing only in the inclusion or exclusion of immediate addressing. The *immediate* instruction set allows instructions to be made up of more than one byte in the RAM where one byte serves and an opcode and subsequent bytes as parameters. In these cases, the program counter is incremented according to the number of bytes used as parameters. Conversely, the *indirect* instruction set allows only single-byte instructions with parameters taken from registers or indirectly accessed RAM addresses. We investigate these two different approaches to see whether corruption of multi-byte instructions during the breeding step (due to splitting and recombination of program strings) will have an effect on the evolutionary process.

The instruction sets are designed such that all possible 8-bit values are mapped to an instruction. As a result any byte array with the aggregate size of all the aforementioned segments may constitute a genetic string. Since such an array fully represents the state of the VM, it ensures that interpreting the same genetic string any number of times will always produce the same result. This allows us to save or recall any genetic string the system generates and reinterpret it later for more detailed analysis.

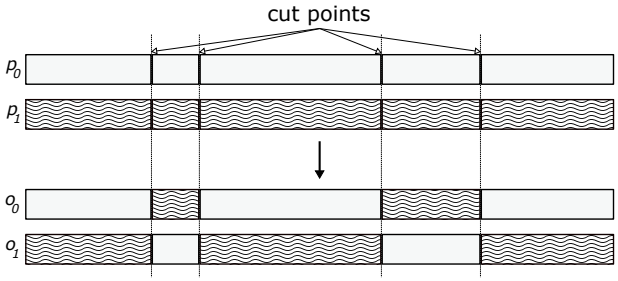


Fig. 6: The crossover process generates random cut points and splices the genetic strings of parents p_0 and p_1 to create two offspring o_0 and o_1.

4 Evolving genetic strings

As mentioned in Section 1, our subject of evolution is the composition process rather than the product. We evolve the genetic strings that represent the initial conditions for this process, while the programs themselves run on a fixed VM architecture [26].

The initial population of genetic strings consists of randomly generated byte arrays of a given size. Because our genetic strings are relatively large, we only keep the current generation of pieces in memory to keep the system efficient. To enable elitism, we employ a generation gap method [32] whereby each subsequent generation is created by allowing both a subset of individuals from the current generation to survive and also by breeding selected pairs together to produce new offspring. The selections for both of these processes are determined randomly from a probability distribution weighted by the fitness test grades of the individuals (see Section 6). A graphical representation of this selection process for survival and crossover can be seen in Figure 5.

4.1 Selection for survival and breeding

When creating generation $n+1$ using the grades from generation n, we begin with survivor selection. Each candidate's survival is determined by their overall grades scaled by a fixed factor F and its age. Aging of individuals was added to prevent elites causing the system to converge too quickly on a sub-optimal result [13]. In our tests F is set to 15% (see Section 7) so an individual with an overall grade of 50% has a 7.5% chance of survival. Each individual is attributed an age of 1 upon creation and this value is increased every time it survives into the next generation. If it reaches a preset maximum age (in the current experiments it is set to 3), it is no longer eligible to be a survivor in the se-

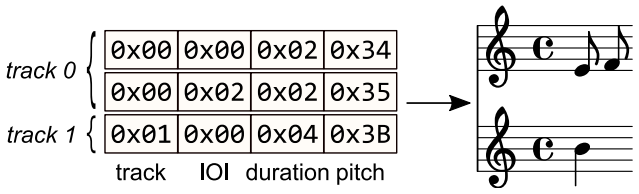


Fig. 7: The model parser turns a byte array into a musical model by separating it into 4-byte chunks and using the values of those bytes as parameters for a note that will be added to the model. The first byte denotes the track index and the next three are the note parameters IOI, duration and pitch.

lection process. This probabilistic approach to survival differs from our previous research [34], where a preset percentage of highest scoring candidates were chosen in each generation. The impact of this change is discussed in Section 8.3.

The selection of parents for crossover is determined using *complementary phenotype selection* [7]. In this process, 'mothers' are chosen based on roulette-wheel selection then hypothetical best-case offspring are created by taking the maximum of each potential parents' grades on each test. The 'fathers' are then chosen through roulette-wheel selection on these hypothetical children rather than on the fitness of the candidates themselves. This method exploits the multi-dimensional nature of our fitness criteria [12], encouraging diversity by assigning high probability to the mating of parents who score highly on different tests.

4.2 Crossover & mutation

When creating and breeding genetic strings, we do not concern ourselves with how their underlying data will be used later because, for the purposes of crossover and mutation, they can simply be viewed as byte arrays. We will refer to the genetic strings of any two parents as p_0 and p_1. With p_0 and p_1 chosen, the *genetic string breeder* builds two new offspring: strings o_0 and o_1. To perform crossover, it chooses a number of cut points, separates p_0 and p_1 into chunks and populates the offspring by inserting alternating sections from each parent (as shown in Figure 6). Children o_0 and o_1 are then mutated by taking a number of random byte indices, and randomizing the content of those bytes. The ratios between the size of the genetic string and the maximum numbers of cut points and mutated bytes are predefined global parameters for each run of the experiment.

5 Representing music

Our phenotype is a musical model that represents a composition as a set of tracks, each consisting of a set of notes. Each note has the following properties:

1. *Inter-onset interval (IOI)* - The time period between the onset of the previous note and the current note in a particular track. For the first note in a track, it is the time interval between the beginning of the piece and the note's onset. Using the same conventions as standard MIDI[2] files [33], the unit of measurement for this property (called a tick) is mapped to sheet music time using a global property `TPQ` *ticks per quarter note*, mapped to real-world time using the property `QPM` *quarter notes per minute*. In our case, we use `TPQ` $= 4$ (i.e. the unit is a 16th note) and `QPM` $= 120$.
2. *Duration* - Time period between the onset and offset of the note in the same unit of measurement as the IOI.
3. *Pitch* - A 7-bit numeric value (between 0 and 127) representing pitch as defined in the MIDI protocol. The value 69 is associated with the 440Hz concert A, with an increase or decrease of one unit representing a one semitone rise or fall in pitch respectively.

Figure 7 shows how the *model parser* interprets the output byte array from the VM to produce a musical model. There are many ways in which this byte stream could be parsed to translate it to music, indeed the output bytes from the VM could be masked in such a way as to produce valid MIDI data directly. However, the MIDI format treats note on and note off events as separate atomic entities in time so this approach would require the evolutionary algorithm to evolve a solution for producing associated pairs of events before it could evolve higher level musical structures. For this reason, we have chosen the current arrangement so that all the bytes that define an individual note event are stored in one place in the byte array. The model parser segments the array into 4-byte chunks, each interpreted as properties of a note (i.e. track, IOI, duration, and pitch). It can also perform bit masking for each of these bytes to constrain their values. These masks, as well as the number of tracks, are global properties of the model parser. In our test cases, we constrain the IOI and duration of each note to a maximum of 16 (i.e. the duration of a whole note) and the pitch to maximum 127 (to comply with the MIDI standard as described above). The number of tracks here is determined by the corpus we use for similarity tests (as detailed in Section 6).

[2] Musical Instrument Digital Interface

6 Assessing musical quality

The fitness of a musical model is determined by assessing statistical similarities to a corpus of real music. In our current experiments, we have chosen Bach's *Inventions and Sinfonias*[3] comprising thirty keyboard exercises. This corpus was chosen because it contains relatively short pieces of similar lengths (around 90 seconds); and also because of its stylistic and structural homogeneity, which we hoped would help narrow the solution space to a certain extent.

The files in the corpus have been generated from a score rather then recorded by a human player. As a result they contain no elements of musical expression such as microtiming variation or changes in dynamics and tempo. All notes have velocity 127, therefore our current system does not use velocity as a note property. This is a deliberate decision in the design of the fitness test because the system produces score-level representations of music with the intention of evolving musical traits such as harmony, melody and rhythm. Musical expression and performance interpretation are not intended outcomes of the current work and, as such, would effectively become noise in the fitness evaluation if they were present in the corpus.

We use two versions of the corpus in our experiments: single- and dual-track. Both contain the same pieces, each comprising the same set of notes, but the dual-track versions are separated into two voices divided by pitch range (effectively splitting the left and right hand keyboard parts). Using the same corpus of music in a multi-track context allows us to evaluate the impact of varying track numbers without introducing stylistic changes that would result from using alternate corpora. For each experiment we keep the number of tracks in a corpus constant and restrict the model parser to producing pieces with that number of tracks.

6.1 Normal distribution tests

For our first type of test, we evaluate two single-value properties of a model: the *total duration* and *number of notes per track*.

To test similarity, we assume a normal distribution for both properties. We calculate the mean μ_c and standard deviation σ_c of all lengths and number of notes in the corpus. The normal distribution is scaled so the value of μ_c gives a fitness of 1. This results in the first two tests within our similarity test container. Figure 8 shows the deduced normal of the length test and the histogram of the lengths of the corpus, to show the correlation.

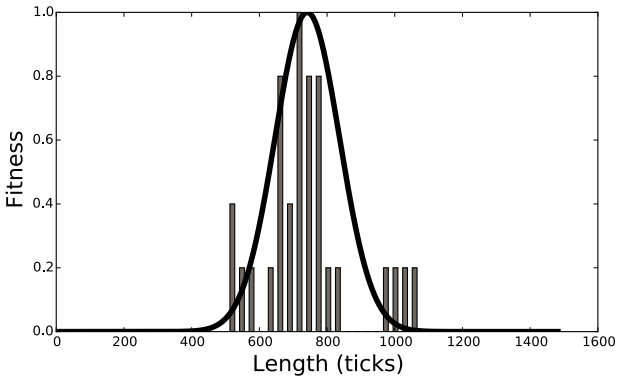


Fig. 8: Deduced normal distribution test for musical model length. The length distribution for the pieces in the corpus (histogram bars) determines the grading schema we use for input models. The thick overlaid line shows the deduced normal curve.

6.2 Descriptor correlation tests

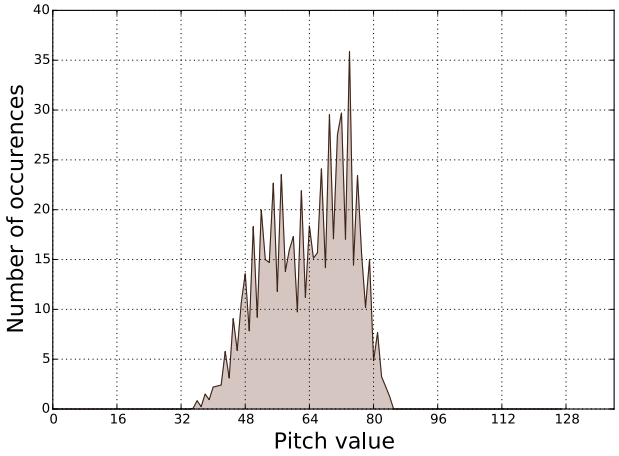
In this system, a *descriptor* is the output of a series of transform methods applied to an input model. For the remaining tests we calculate four transforms applied to each of the three note properties (IOI, duration and pitch) in each track resulting in a total of twelve statistics. The following transforms are used to build the descriptor:

Histogram A histogram shows the distribution of the different values within the input data. For example, when analysing inter-onset intervals, a histogram represents the distribution of quarter notes and eighth notes etc. If two descriptors have a similar histogram for a property, it means the distribution of that property is similar (An example is shown in Figure 9). Selecting for this discourages unusual values (e.g. very low or very high pitches).

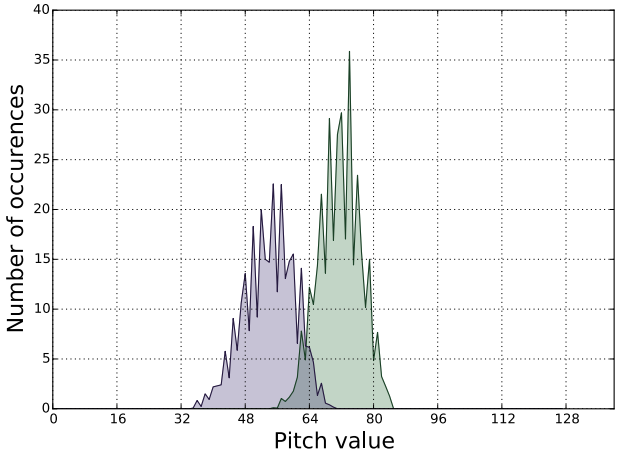
Histogram of differential This transform shows the distribution of the rate of change between consecutive notes. It represents the contour of a property. Selecting for this discourages unlikely changes (e.g. large rises or falls in pitch).

Fourier transform The histograms discard time information, therefore they are not appropriate for testing repetition and structure. By applying a discrete Fourier transform to a property, we can see repetitive time-domain patterns appear as peaks in the spectrum (An example of this for IOI is shown in Figure 10).

[3] Works BWV 772-801, MIDI files for which were downloaded from `www.midiworld.com/bach.htm`



(a) Single-track corpus



(b) Dual-track corpora: the green area on the right and the purple one on the left represent the first and second tracks, respectively.

Fig. 9: The mean pitch histogram of all members of the single and dual-track corpora. The single-track pitch distribution is the sum of the two tracks of the dual-track corpus since they comprise the same notes. A model scores high on the associated test if its pitch distributions for both tracks are similar to those from the corpus.

Selecting for these properties may encourage the emergence of repetitive patterns and rhythm.

Fourier transform of differential This transform shows the repeating patterns in the rate of change between consecutive values again selecting for repetitive patterns in a note property through the piece.

All transforms give results 128 bins in width, independent of the input size. The histograms measure discrete values from 0 to 127; each note property lies within these boundaries due to bit masking (see Section 5). The histogram of the differential measures the signed values of changes from -64 to 63. The Fourier transforms are performed at a fixed size of 2048 points (sufficient for the longest possible output pieces) then downsampled to fit the 128-bin descriptor. This results in a matrix with 128 columns and $4vk$ rows, where v is the number of tracks and k is the number of properties (in this case, $k = 3$).

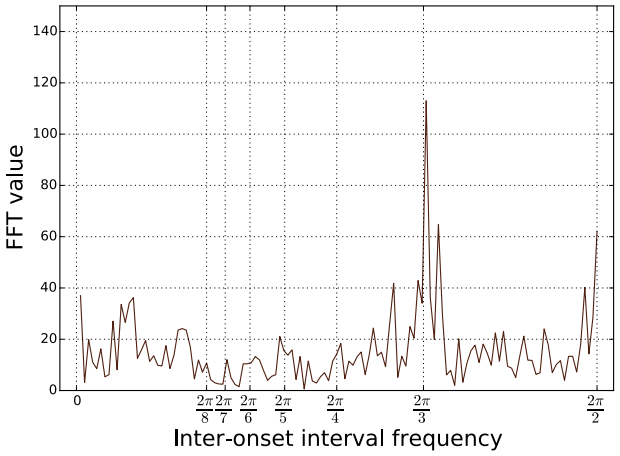


Fig. 10: The inter-onset interval spectrum of the first member of the single track corpus. The peaks at frequencies $\omega = \pi$ and $\omega = \frac{2\pi}{3}$ show strong repetition of motifs comprised of 2 and 3 notes, respectively. Models showing similar patterns of repetition score high on the similarity test associated with this property.

Comparison of two descriptors is performed using the *Pearson correlation coefficient* [31]. The correlation r is measured row-by-row therefore different weights can be assigned to the correlation of different properties to determine their relative importance. The results are averaged across tracks, resulting in $4k = 12$ grades.

Given the ith rows from descriptors D and E, denoted by D_i and E_i, and their respective mean values $\bar{D}_i$ and $\bar{E}_i$, their correlation coefficient is given by the following equation:

$$r_{D_i,E_i} = \frac{\sum_{i=0}^{127}(D_i - \bar{D}_i)(E_i - \bar{E}_i)}{\sqrt{\sum_{i=0}^{127}(D_i - \bar{D}_i)^2}\sqrt{\sum_{i=0}^{127}(E_i - \bar{E}_i)^2}} \quad (1)$$

The resulting coefficient is a value between -1 and 1; positive values implying positive correlation, negative values negative correlation and a value of 0 implies no correlation. To obtain a fitness test grade, we rectify r_{D_i,E_i} returning 0 to remove negative fitness values. This approach allows us to return grades between the values of 0 and 1 without having to normalize the descriptors themselves.

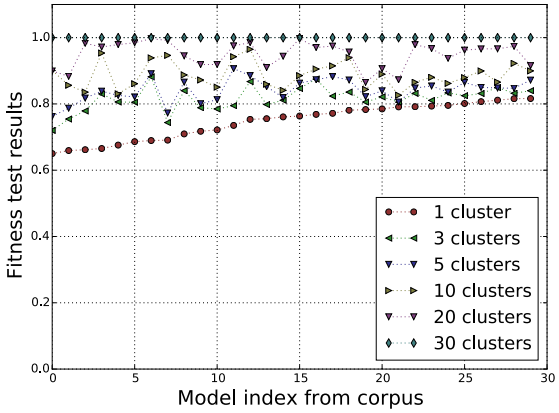


Fig. 11: Similarity test results for the corpus using different numbers of clusters; the models have been sorted by correlation.

6.3 Corpus clustering

We have defined correlation between two individual descriptors but our corpus consists of models for thirty different pieces, all of which have a different descriptor. As a result, a method must be used to incorporate the data from all thirty corpus members into the test. A possible reference descriptor could be a conceptual 'average' of all descriptors from the corpus. This would be a *single niche* within our solution space [22]. We experimented with this approach by testing the members of corpus against such a descriptor. We found this to give relatively low grades (between 65 and 82%) which suggested that taking just one niche narrows down our solution space so much that even the corpus members themselves cannot achieve a high enough grade.

An alternative approach is to use all corpus members as niches, i.e. measure an input model's correlation to each of the corpus members and take the maximum value. In this case, all corpus members score 100%. However, doing this many comparisons for every generated piece would be computationally expensive and might also broaden the solution space too much. The latter issue could be addressed by using gamma correction on the correlation results (see Section 9).

To obtain the best of both worlds, we *cluster* our corpus, partitioning the descriptors into K subsets. Afterwards we can use the centres of these clusters as reference descriptors, and test for the maximal correlation with each centre.

We use *k-means clustering* [15], which is performed once offline before the evolutionary algorithm starts. Its steps are as follows:

1. Define the cluster centres as the first K descriptors.
2. Classify each descriptor into one of the K partitions by taking the minimal square error.
3. Find the new centres of gravity by taking the mean descriptor of each partition. Assign these as the new cluster centres.
4. Repeat steps 2-3 until no change occurs between iterations, or the number of steps reaches a maximum value (to avoid infinite oscillation).

This approach allows the flexibility of changing the value of K between runs. The two methods proposed initially can also be achieved by setting $K = 1$ or $K = 30$ respectively. Figure 11 shows the grades achieved for the corpus using different values for K.

7 Experiments

Our experiments explore the space of possible system parameters to measure their impact on the progress of the evolutionary process. Our aim is to investigate which set of parameters gives the highest grades consistently and determine which parameters help the grades represent musical quality most accurately.

For each set of parameters, we have executed twenty separate test runs, each time allowing the system to complete 20,000 generations. Parameters kept constant for all runs are a survival probability of 15%, maximum survival age of 3, maximum cut point ratio of 0.1%, and maximum mutation ratio of 2%. We used 5 corpus clusters and VM halting conditions of completing 60,000 fetch cycles or producing 2,600 output bytes. The following parameters were varied between trials:

1. Population size $N \in \{2^x : 4 \leq x \leq 10\}$
2. Number of tracks in the corpus $v \in [1, 2]$ (discussed in Section 6);
3. VM instruction set; either immediate or indirect (discussed in Section 3).

This results in a total of 28 possible system configurations. With the exception of our new survival mechanism (detailed in Section 8.3), the parameter set $N = 256$, $v = 1$ and the immediate addressed instruction set is directly equivalent to the system described in our previous work [34].

The fitness values for each generation are recorded while the genetic strings for the whole population are exported to disk every 1000 generations for later analysis. The highest scoring models in the saved generations are also exported as MIDI for subjective assessment of the results.

Table 1: Mean and maximum grades for each set of tested parameters. Larger population sizes give better grades overall; dual-track tests have higher means but lower maxima. There is no significant difference between the immediate and indirect instruction sets for the VM.

	Immediate instruction set				Indirect instruction set			
	Single-track		Dual-track		Single-track		Dual-track	
Population size N	Mean	Max	Mean	Max	Mean	Max	Mean	Max
16	0.385	0.489	0.352	0.455	0.379	0.483	0.360	0.479
32	0.435	0.580	0.411	0.550	0.424	0.532	0.415	0.598
64	0.446	0.602	0.429	0.616	0.476	0.608	0.431	0.636
128	0.482	0.648	0.429	0.654	0.474	0.583	0.432	0.609
256	0.483	0.662	0.450	0.689	0.476	0.628	0.459	0.640
512	0.492	0.679	0.471	0.706	0.502	0.626	0.476	0.673
1024	0.504	0.667	0.490	0.757	0.513	0.641	0.514	0.732

8 Results

Table 1 shows the mean and maximum grades of the final generations in each test run. Analyzing the results, we can draw some general conclusions: Larger population sizes result in both better mean and maximum grades; using the dual-track corpus results in smaller mean values for the populi, but larger maxima; and the inclusion of immediate addressing to the VM instruction set seems to make no real difference in the results (the mean and maximum values show an average difference of 0.5% and 2% respectively).

8.1 Using different population sizes

We tested population sizes increasing in powers of two from 16 to 1024. The experiments show better grades emerging from larger population sizes. Even the two largest population sizes (512 and 1024) show a clear difference with increases of 2.8% and 2% in maximum and mean grades, respectively. However, runtime grows linearly with population size so an exponential increase in population size likewise causes an exponential increase in runtime, taking the time required for a run from hours to days for much larger populations. Using the data points that we have, we can estimate if increasing the population size further would continue to yield significantly better results. Figure 12 shows the results

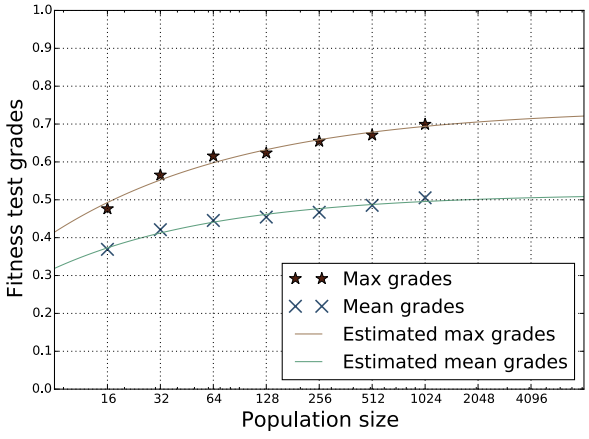


Fig. 12: Maximum and mean grades at the final generation, averaged over all test runs. The lines show an estimated curve fit over these samples, suggesting that further increasing the population size would not give significantly better results.

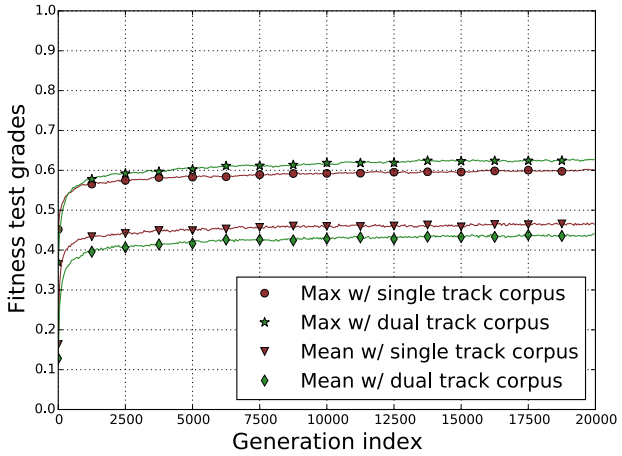


Fig. 13: Mean and maximum grade progression over the generations when using single and dual-track corpus. The dual-track corpus shows a slightly higher maximum, but lower mean.

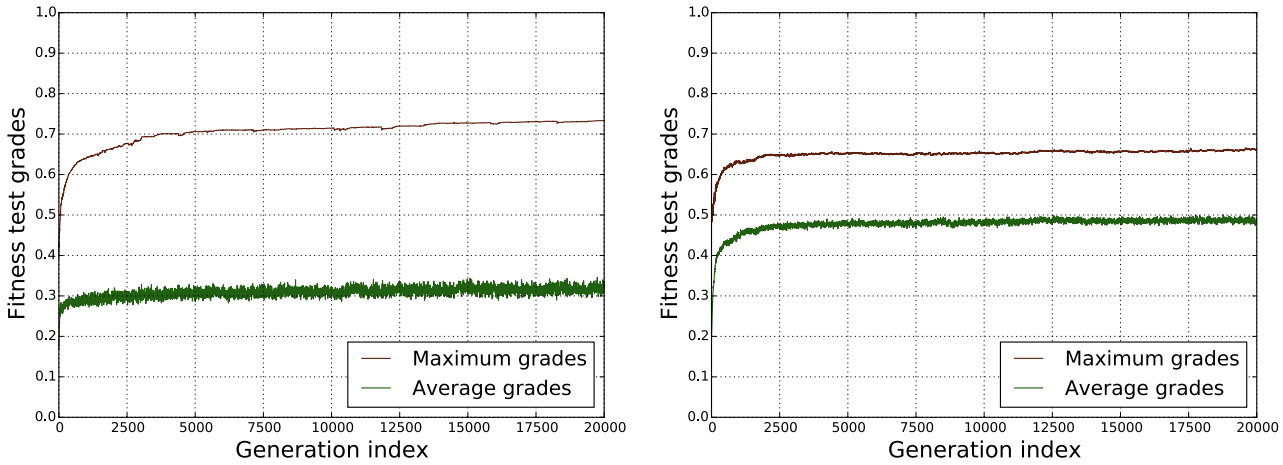


Fig. 14: Comparison of mean/maximum grade progression for different survival mechanisms. The left plot shows results from our experiment in [34] ($N = 256$, $v = 1$, immediate addressing) with guaranteed survival of the highest scoring candidates. The right plot shows results from our current experiment run with the same configuration but using probabilistic survival. The results show a slightly smaller maximum but a much larger mean, suggesting that probabilistic survival allows a more diverse population to emerge at the cost of not allowing strong but fragile individuals to survive.

in Table 1 averaged over all configuration dimensions with the exception of population size. Using linear regression we attempt to fit a curve to these points for a function

$$f_{\alpha,m}(N) = m \times (1 - \alpha^{-\log_2 N}) \quad (2)$$

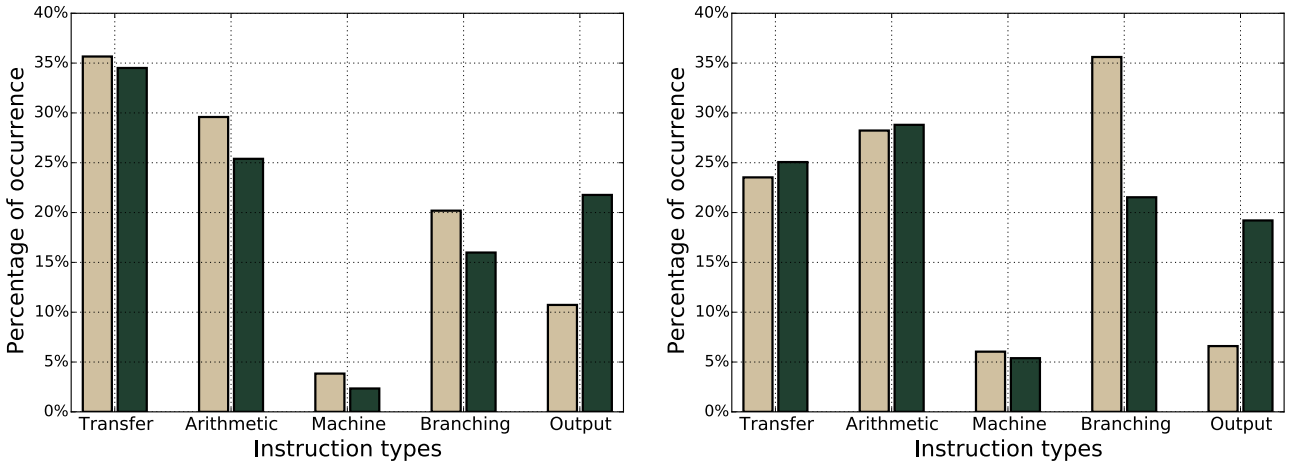
where α impacts the slope of the exponential curve and m represents the hypothetical highest grade we can achieve in the system. Fitting $f_{\alpha,m}$ separately for the mean and maximum grades gives acceptably small errors: averages of 0.01 and 0.006 respectively. Extrapolating on the fitted curves suggests that the small gains to be made by further doubling the population size will result in only marginally better results while requiring significantly more computing resources.

8.2 Single vs. dual-track corpus

As mentioned in Section 6, we have performed separate experiments using single and dual-track versions of the test corpus. Figure 13 shows the resulting grade progressions averaged over all dimensions except the number of tracks. The dual-track corpus shows an improvement in results with slightly larger maxima than the single-track. The rate of fitness growth through the generations are almost identical, suggesting that the number of tracks does not impact the speed at which pieces are able to evolve. While the maxima for dual-track increased the means fell which may be a result of the descriptor size being proportional to the number of tracks. When comparing two descriptors in the dual-track case, we are assessing correlation of data twice as large and this might be expected to make it more difficult for high correlations to emerge.

8.3 Survival mechanism

As mentioned in Section 4.1, in contrast with our experiments from [34], candidate survival here is based on fitness and chance rather than guaranteed for the highest scoring individuals. Figure 14 shows the side-by-side comparison of the guaranteed versus probabilistic survival mechanisms for otherwise identical parameters. Both plots show the progression of the maximum and mean grades over 20,000 generations, averaged over all test runs. The maximum values for the probabilistic survival strategy are lower (the values in the last generation drop from 0.733 to 0.662) but the mean grades are much higher (rising from 0.324 to 0.482). This may be because guaranteed survival tends to single out one or more marginally favourable candidates, who are always propagated into the next generation. However, since genetic strings can be fragile when faced with crossover and/or mutation (even a small change may drastically alter the resulting process executing on the VM), guaranteed survival does not allow as much population diversity.



(a) Immediate OCI

(b) Indirect OCI

Fig. 15: Rate of occurrences of different instruction types encountered by the VM during execution. The left bar in each column shows the percentage of a specific instruction type when using random genetic strings (as in generation zero) while the right bar shows the mean percentages of all highest scoring individuals in our tests. We can conclude that the algorithm encourages the emergence of more output instructions and less branching.

8.4 VM instruction type occurrences

As discussed in Section 3, we can group the VM's instruction set into categories. We observed that the rate of occurrence of different instruction types encountered by the VM during execution changed over the generations, as shown in Figure 15.

The results in both instruction sets suggest the algorithm favours more output instructions than occur in random data; both percentages have at least doubled during the run. This shows that too few output instructions occur when evaluating a random genetic string, prohibiting the individuals to achieve high grades on the test related to the number of notes (see Section 6.1). We can also observe a decline in the number of branching instructions which may be due to the detrimental effect of infinite loops.

8.5 Subjective evaluation

Listening to the resulting MIDI files allows us to subjectively evaluate the musicality of the results. Their durations are almost universally within the boundaries dictated by the corpus statistics, although many achieve this with a smaller than desired number of notes (i.e. there may be very long notes or rests).

Some musical traits emerge in the results, especially related to repetition. Almost all the MIDI files contain motifs of a few notes repeated multiple times, and some show variation on the repeated theme. The system successfully finds small motifs consisting of a few notes which, when repeated many times, allow the result to approach the statistics of the corpus. However, these results are not particularly musical (see example in Figure 16), suggesting the need for further fitness tests that can limit such 'shortcuts' to high grades.

Although repetition and variation emerge, other musical properties such as *harmony*, *melody* or *entropy* are somewhat lacking. Adding further fitness tests, perhaps some inspired by music-theory, may help the system to find solutions that exhibit such traits.

On the whole, the multi-track results seem more musical than the single-track examples. This may be because both tracks can exhibit small repeating patterns, which sometimes have different lengths resulting in interesting rhythms when they are played together (see Figure 17).

All the MIDI files and grades generated by the system are available at the project web page[4]. The algorithm has produced a few results with more subjectively pleasing motifs than those shown in Figures 16 and 17. However, such examples were rare exceptions for which our fitness tests may not be able to take full credit, hence the excerpts shown here were felt to be more representative of the overall results.

[4] `http://csabasulyok.bitbucket.org/emc`

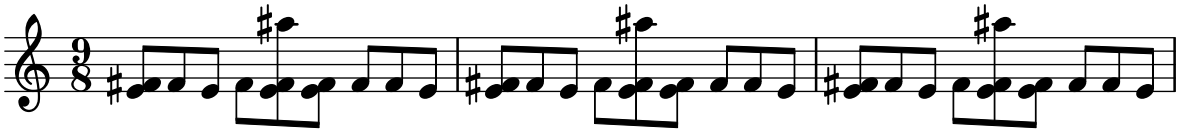


Fig. 16: A single track result. It shows a small repeating pattern over the entire duration of the piece. The time signature has been added manually to visualize the pattern.

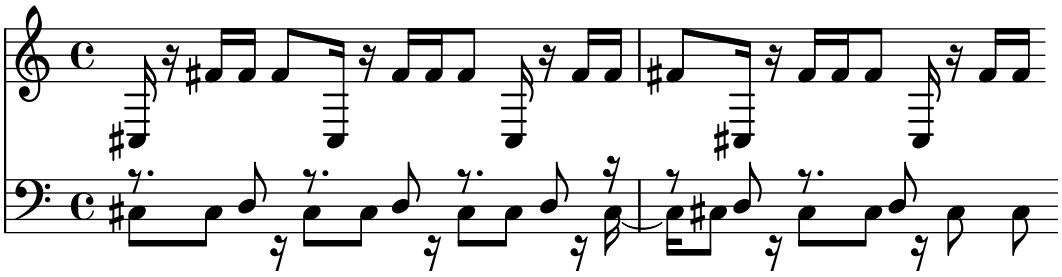


Fig. 17: A dual-track result, where two different repeating patterns emerge on the two tracks. The motifs have a different length (6 and 5 ticks respectively), resulting in an interesting rhythmic pattern when overlaid.

9 Conclusions and future work

The space of possible music is vast, even when analysing only a few properties and ignoring dimensions such as sound synthesis and performance as we have done here. Despite applying many constraints there is no single, universal location of a 'subspace of good music'; it is a subjective evaluation. Results from the literature show promise however, and many real-world musical pieces have been composed with the assistance of evolutionary algorithms. Indeed, Waschka [39] has always viewed these algorithms as auxiliary tools for composer inspiration instead of standalone virtual composers.

In this paper we have demonstrated a novel approach to evolutionary music composition: evolving the composition process rather than the product. We have incorporated elements of linear genetic programming and a novel approach to fitness test design using musical corpora. By using a Turing-complete VM, we have successfully modelled the composer; the programs to be run on the VM represent the genotypes and the resulting output musical pieces the phenotypes. Separating the program structure from the musical output in this way has allowed us to model the search space indirectly via the instruction set of the VM.

Unlike many previous approaches in the literature, we have deliberately avoided narrowing the search space, either by artificially constraining parameters (e.g. using only pitches within a diatonic scale) or by setting favourable initial conditions (e.g. using the corpus as the starting population). Instead, we give the virtual composer complete freedom then attempt to guide the search using our corpus-based fitness tests. By providing a set of niches (the clustered corpus descriptors), we give the system a measure by which it may deem musical outputs to be 'good' if they converge on a descriptor with similar statistics.

We have explored the space of possible system parameters and concluded that 1024 individuals is an optimal population size in the tradeoff between evolutionary progress and computational cost of the system. We found the presence or absence of immediate addressing in the VM instruction set did not influence the results, while using a multi-track context gave a more detailed description of the corpus and therefore helped more interesting pieces to emerge.

Our results are promising, demonstrating that while this approach succeeds at converging towards chosen properties of the pieces in the corpus, it still only produces partially musical results. The system often achieves high grades with very small motifs repeated many times. While these motifs can readily be used to inspire a human composer, achieving completely machine-driven composition will require further fitness tests that select for higher-level structures in the output. For example, introducing tests on properties such as entropy could potentially force the evolutionary search to discard these overly repetitive results in favour of more complex, musically interesting outcomes. Likewise, the current musical results can be quite dissonant and generally lack a stable metrical structure. By introducing tests based on the statistics of harmonic interval rela-

tions over time rather than simply using pitch distribution information we may potentially encourage elements of mode and harmonic progression to emerge. It is also the case that our current representation of the musical model is in function of note index, not time, which limits the emergence of rhythm and metre. Transforming a model to a function of time would allow the inclusion of new tests for such properties. In a multi-track context, our descriptor correlation tests only measure the similarities per track. Tests which measure overall statistical similarity might combine the strengths of the single- and multi-track approaches.

Improvements could also be made by altering our choice of corpus. The versions of the corpus pieces we have used here have no dynamics recorded so there is currently no frame of reference to allow this kind of musical expression to emerge. It would therefore be interesting to explore what musical pieces played by human musicians could teach our system to do if we incorporated tests for elements of expressive performance such as note velocity or tempo variation.

Our current approach to comparing a musical model to the corpus (clustering its members and taking the maximum correlation) prohibits the emergence of pieces truly similar to the corpus members since even they do not achieve perfect grades. A possible alternative method would be to use gamma correction in the correlation tests allowing the incorporation of all corpus members without overly broadening the solution space.

In this work we have investigated two alternative instruction sets for our VM but its structure has so far been kept constant. Varying aspects of the VM structure such as the size of main memory or processor architecture and arrangement of stack and registers is an important next step for our experiments. Memory size was set to 64kB in our present system because this is the space addressable with 16-bit registers. Given the tendency for short repetitive motifs in our output pieces, it would be interesting to analyse how much of this memory is actually being used during the execution of the genetic programs and what effect we would see by constraining the available space. Another avenue that would be interesting to explore would be alternate VM architectures. The VM is an emulation so there is no requirement to keep to the sequential von Neumann model which is potentially not very robust to crossover and mutation. For this reason, we may also investigate tree-based functional approaches common to classical genetic programming or more parallel architectures such as programmable systolic arrays in future experiments.

References

1. Alfonseca, M., Cebrian, M., Ortega, A.: A simple genetic algorithm for music generation by means of algorithmic information theory. In: IEEE Congress on Evolutionary Computation, pp. 3035–3042. IEEE (2007)
2. Biles, J.: GenJam: A genetic algorithm for generating jazz solos. In: Proceedings of the 1994 International Computer Music Conference, ICMA, pp. 131–137 (1994)
3. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming, 1st edn. Springer Publishing Company, Incorporated (2010)
4. Chen, C.C.J., Miikkulainen, R.: Creating melodies with evolving recurrent neural networks. In: Proceedings of the INNS-IEEE International Joint Conference on Neural Networks, pp. 2241–2246. IEEE, Piscataway, NJ (2001). URL `http://nn.cs.utexas.edu/?chen:ijcnn01`
5. Dahlstedt, P.: Autonomous Evolution of Complete Piano Pieces and Performances. In: 9th European Conference on Artificial Life (2007)
6. De Prisco, R., Zaccagnino, G., Zaccagnino, R.: A multi-objective differential evolution algorithm for 4-voice compositions. In: Differential Evolution (SDE), 2011 IEEE Symposium on, pp. 1–8 (2011). DOI 10.1109/SDE.2011.5952053
7. Dolin, B., Arenas, M.G., Guervós, J.J.M.: Opposites attract: Complementary phenotype selection for crossover in genetic programming. In: Proceedings of the 7th International Conference on Parallel Problem Solving from Nature, PPSN VII, pp. 142–152. Springer-Verlag, London, UK, UK (2002). URL `http://dl.acm.org/citation.cfm?id=645826.669282`
8. Donnelly, P., Sheppard, J.: Evolving four-part harmony using genetic algorithms. In: Applications of Evolutionary Computation, vol. 6625, pp. 273–282. Springer Berlin Heidelberg (2011)
9. Dostál, M.: Musically meaningful fitness and mutation for autonomous evolution of rhythm accompaniment. Soft Computing **16**(12), 2009–2026 (2012). DOI 10.1007/s00500-012-0875-8. URL `http://dx.doi.org/10.1007/s00500-012-0875-8`
10. Eigenfeldt, A.: The evolution of evolutionary software: Intelligent rhythm generation in kinetic engine. In: EvoWorkshops, vol. 5484, pp. 498–507. Springer (2009)
11. Eigenfeldt, A.: Corpus-based recombinant composition using a genetic algorithm. In: Soft Computing, vol. 16, pp. 2049–2056. Springer (2012)
12. Fonseca, C.M., Fleming, P.J.: Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 416–423. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993). URL `http://dl.acm.org/citation.cfm?id=645513.657757`
13. Ghosh, A., Tsutsui, S., Tanaka, H.: Individual aging in genetic algorithms. In: Intelligent Information Systems, 1996., Australian and New Zealand Conference on, pp. 276–279. IEEE (1996)
14. Gibson, P., Byrne, J.: Neurogen, musical composition using genetic algorithms and cooperating neural networks. In: Artificial Neural Networks, 1991., Second International Conference on, pp. 309–313 (1991)
15. Hartigan, J.A.: Clustering Algorithms, 99th edn. John Wiley & Sons, Inc., New York, NY, USA (1975)
16. Hartmann, P.: Natural selection of musical identities. In: International Computer Music Conference (1990)

17. Horner, A., Goldberg, D.E.: Genetic algorithms and computer-assisted music composition. In: R.K. Belew, L.B. Booker (eds.) ICGA, pp. 437–441. Morgan Kaufmann (1991)
18. Horowitz, D.: Generating rhythms with genetic algorithms. In: AAAI, vol. 94, p. 1459 (1994)
19. Jacob, B.: Composing with genetic algorithms. In: International Computer Music Association, pp. 452–455 (1995)
20. Loughran, R., McDermott, J., O'Neill, M.: Grammatical evolution with Zipf's law based fitness for melodic composition. In: J. Timoney (ed.) Proceedings of The 12th Sound and Music Computing Conference. Maynooth, Ireland (2015)
21. MacCallum, R.M., Mauch, M., Burt, A., Leroi, A.M.: Evolution of music by public choice. Proceedings of the National Academy of Sciences **109**(30), 12,081–12,086 (2012)
22. Mahfoud, S.W.: Niching methods for genetic algorithms. Tech. rep. (1995)
23. Martins, J.M., Miranda, E.R.: Emergent rhythmic phrases in an A-Life environment. In: Proceedings of ECAL 2007 Workshop on Music and Artificial Life (MusicAL 2007), pp. 11–14 (2007)
24. McIntyre, R.: Bach in a box: the evolution of four part baroque harmony using the genetic algorithm. In: Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, pp. 852–857 vol.2 (1994). DOI 10.1109/ICEC.1994.349943
25. Miranda, E.R., Biles, J.A.: Evolutionary Computer Music. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
26. Nordin, P., Banzhaf, W.: Evolving Turing-complete programs for a register machine with self-modifying code. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 318–327. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995). URL http://dl.acm.org/citation.cfm?id=645514.657920
27. Nuanáin, C.Ó., Herrera, P., Jorda, S.: Target-based rhythmic pattern generation and variation with genetic algorithms. In: Proceedings of The 12th Sound and Music Computing Conference. Maynooth, Ireland (2015)
28. Phon-Amnuaisuk, S., Tuson, A., Wiggins, G.: Evolving musical harmonisation. In: In ICANNCA (1999)
29. Reddin, J., McDermott, J., O'Neill, M.: Elevated pitch: Automated grammatical evolution of short compositions. In: Applications of Evolutionary Computing, *Lecture Notes in Computer Science*, vol. 5484, pp. 579–584. Springer Berlin Heidelberg (2009)
30. Rodriguez, J.D.F., Vico, F.J.: AI methods in algorithmic composition: A comprehensive survey. Journal of Artificial Intelligence Research pp. 513–582 (2013)
31. Rummel, R.J.: Understanding correlation. Honolulu: Department of Political Science, University of Hawaii (1976)
32. Sarma, J., De Jong, K.: Selection: generation gap methods. In: Handbook on Evolutionary Computation, pp. C2.7:1–C2.7:5. Institute of Physics Publishing and Oxford University Press, Bristol and New York (1997)
33. Stansifer, R.: The MIDI File Format. http://cs.fit.edu/~ryan/cse4051/projects/midi/midi.html
34. Sulyok, C., McPherson, A., Harte, C.: Corpus-taught evolutionary music composition. In: Proceedings of the 13th European Conference on Artificial Life, pp. 587–594. York, UK (2015)
35. Thywissen, K.: GeNotator: An environment for exploring the application of evolutionary techniques in computer-assisted composition. Org. Sound **4**(2), 127–133 (1999). DOI 10.1017/S1355771899002095. URL http://dx.doi.org/10.1017/S1355771899002095
36. Tokui, N., Iba, H.: Music composition with interactive evolutionary computation. In: Proceedings of the 3rd international conference on generative art, vol. 17, pp. 215–226 (2000)
37. Towsey, M., Brown, A., Wright, S., Diederich, J.: Towards melodic extension using genetic algorithms. Educational Technology & Society **4**(2) (2001)
38. Unemi, T.: SBEAT3: A tool for multi-part music composition by simulated breeding (2002)
39. Waschka, R.I.: Composing with genetic algorithms: GenDash. In: Evolutionary Computer Music. Springer London (2007)
40. Whorley, R.P., Rhodes, C., Wiggins, G., Pearce, M.T.: Harmonising melodies: Why do we add the bass line first? In: International Conference on Computational Creativity, pp. 79–86. Sydney (2013). URL http://research.gold.ac.uk/9795/