DOI: http://dx.doi.org/10.7551/978-0-262-33027-5-ch102

Corpus-taught Evolutionary Music Composition

Csaba Sulyok[1], Andrew McPherson[1], Christopher Harte[2]

[1]Queen Mary University of London
[2]University of York
csaba.sulyok@gmail.com

Abstract

In this paper we present a music composition system that uses a corpus-based multi-objective evolutionary algorithm. We model the composition process using a Turing-complete virtual register machine to render musical models. These are evaluated using a series of fitness tests, which judge the statistical similarity of the model against a corpus of real music. We demonstrate that the methodology succeeds in creating pieces of music that converge towards the properties of the chosen corpus. These pieces exhibit certain musical qualities (repetition and variation) not specifically targeted by our fitness tests; they emerge solely based on the similarities.

Introduction

The process of creating and assessing music is fundamentally subjective and hard to define. Composers spend years perfecting and evolving their technique; to create new music successfully, both experience of the composition process and knowledge of existing 'good' music is required. The composers 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.

Since the development of a composer's skill can be viewed as an evolving process, it seems intuitive that evolutionary computation techniques should find utility in algorithmic music composition. Using evolutionary models for composing music is nothing new; examples date back to the early 1990s (Hartmann, 1990; Gibson and Byrne, 1991; Horner and Goldberg, 1991; Biles, 1994; McIntyre, 1994). However, the subjective nature of music quality makes defining appropriate machine fitness tests difficult (Miranda and Biles, 2007; Waschka, 2007). As a result, much previous research in this area has relied on some degree of human feedback for fitness evaluation (Hartmann (1990); Jacob (1995); Tokui and Iba (2000); for an exhaustive review see Rodriguez and Vico (2014)).

To reduce the otherwise vast solution space of all possible music, many previous approaches have focused on evolving one particular musical property such as rhythm patterns

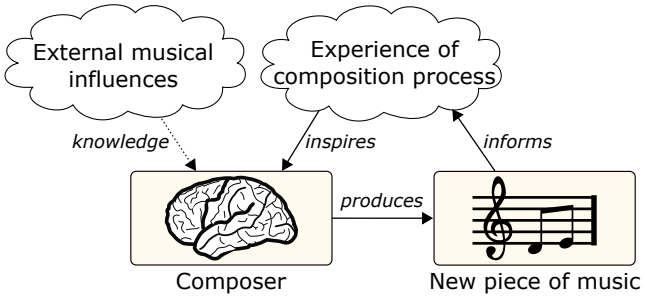

Figure 1: The creative process of a composer is informed by both experience of music in general (external influence of others' music) and experience gained through practice of the composition process itself.

(Tokui and Iba, 2000; Eigenfeldt, 2009; Martins and Miranda, 2007)), melody (Towsey et al., 2001) or harmonisation (Phon-amnuaisuk et al., 1999). Other work looked at generating small musical motifs (*GenDash*, Waschka (2007); *SBEAT*, Unemi (2002)) or evolving improvisatory passages (*GenJam*, Biles (2007)).

In more recent years, evolutionary composition has seen a rise in interest again: De Prisco et al. (2011) used multi-objective differential evolution to solve unfigured bass harmonization; Dostál (2012) explored autonomous fitness tests to evolve rhythm accompaniments; and MacCallum et al. (2012) created the online *DarwinTunes* community to crowd-source human feedback for choosing which music pieces will be selected for breeding.

When designing an evolutionary system, we must first answer the question: "What is this system supposed to evolve?". Previous composition systems have generally attempted to evolve musical pieces but in this paper we propose evolving the composition *process* instead. We cast the action of composing a piece of music as a process running on a Turing-complete virtual computing machine. The virtual machine has a set of instructions that will be executed in a given order depending on the initial state of its memory

Csaba Sulyok, Andrew McPherson, Christopher Harte (2015) Corpus-taught Evolutionary Music Composition. Proceedings of the European Conference on Artificial Life 2015, pp. 587-594

(i.e. its program) and some way of writing notes onto a musical 'score' whenever an output instruction is encountered. The development of the composer's skill then becomes a genetic programming problem; the genotype is the program string presented to the virtual machine and the phenotype its musical output. In such a system, the executing process on the virtual machine can hold internal structuring rules and information that are not visible in the final musical phenotype. Whorley et al. (2013) address this point in the context of melody harmonization, the exact steps of which cannot be known just by listening to the 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 statement in the program can cause the repetition of a single note or section, or perhaps even the entire piece depending on where it occurs. While the musical possibilities of this approach increase in number, the resulting space of possible outcomes is very large and hard to analyse (McIntyre, 1994; Martins and Miranda, 2007) so careful design of fitness evaluation is necessary.

As mentioned above, the composer's creative process cannot operate in a vacuum; it is necessarily dependent on the influence of works from previous composers. In the same way, our evaluation process employs fitness tests that judge the similarity between statistical properties of the current musical phenotype and those of a chosen corpus of existing music (in this case, a group of Bach keyboard exercises). Our evolutionary algorithm is multi-objective in nature, incorporating tests for a number of different properties (Fonseca and Fleming, 1993). Using statistical similarity with a corpus as musical fitness evaluation has been discussed before (Eigenfeldt (2012), Manaris et al. (2007)), but without the inclusion of a Turing-complete virtual machine.

Our aim here is to establish whether the use of virtual machine and corpus-based similarity tests together will help the system converge towards the properties of the chosen corpus. Although the phenotype in our current system is music, we propose this method as a generic one that can be applied to the evolution of any creative process.

System overview

Our system follows a conventional genetic programming structure (see Figure 2). An initial population of genotypes (genetic strings) is generated randomly then each individual is run on the virtual machine in turn. The output of the virtual machine is a byte stream that is parsed by a *model builder* to create our phenotype, a musical model. The structure of this model is completely independent of the structure and mechanism of the virtual machine. This two-stage approach to rendering the model deviates from previous musical evolutionary systems in that the genetic string is not directly used to build the phenotype, instead being *interpreted* by the virtual machine.

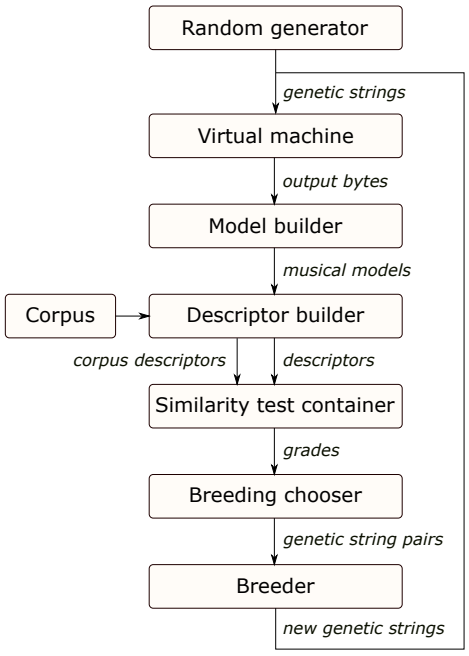

Figure 2: Workflow of the algorithm: A population of genetic strings is interpreted by the virtual machine and the resulting bytes used to build models; statistical transforms are performed on the models to yield descriptors, which are used in similarity tests to assess fitness of the musical pieces. Based on these grades, genetic strings are bred and mutated to produce the next generation.

Each phenotype model is evaluated using a series of tests derived from the corpus. These tests focus on the underlying statistical properties of a model rather than the similarity of the data itself. High grades can therefore be achieved not only when a model is identical to a member of the corpus, but also when it is statistically similar to them. To produce these statistics, we subject models to a series of transform methods to produce a *descriptor*. The construct which creates a descriptor from a musical model is a *descriptor builder*. We refer to the complete set of tests as the *similarity test container*.

Based on the assigned grades, the *breeding chooser* chooses pairs of units for crossover, 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 virtual machine to execute.

Virtual machine

Our Turing-complete virtual machine interprets commands by reading 8-bit values from the current address pointed to by the the program counter and executing the assigned instructions. All possible values of a byte become potential opcodes mapped to a custom instruction set. The following

Csaba Sulyok, Andrew McPherson, Christopher Harte (2015) Corpus-taught Evolutionary Music Composition. Proceedings of the European Conference on Artificial Life 2015, pp. 587-594

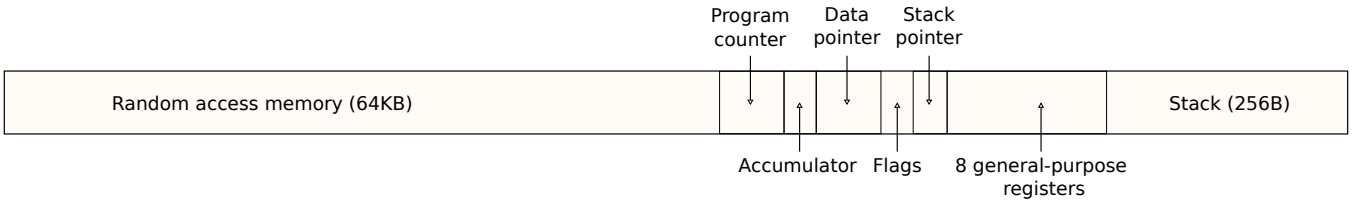

Figure 3: Regions of the genetic string dictate the complete initial conditions of the virtual machine. The first 64kB represents the machine's main memory; eight 8-bit general-purpose registers and an accumulator are provided for data manipulation; a 16-bit program counter and data pointer allow addressing of the main memory; a separate 8-bit stack pointer allows the machine to control a 256-byte call stack.

types of operations are defined:

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/logical functions on a value within the RAM or a register;
3. *Branching & conditional instructions* - move the program counter to a different location, optionally based on a condition;
4. *Machine control* - internal commands such as halting or pushing/popping using the stack;
5. *Output commands* - outputs a value from the RAM or a register; this is the main customization we use to make the virtual machine serve our purpose.

The virtual machine interprets bytes until a 'halt' command is found or until it has processed a preset maximum number of commands or outputs.

The genetic string defines the initial condition of the virtual machine; it encompasses the initial value of the random access memory, the stack and all registers. When a genetic string is fed into the virtual machine, all these segments are set as shown in Figure 3.

Since our virtual machine is a Von-Neumann machine, it can overwrite its own memory while the process executes. To avoid the genetic string being changed in this process, it is always copied rather than moved in the feeding process. This ensures that interpreting the same genetic string multiple times will always result in the same output. Therefore we can safely save, recall or re-evaluate any genotype.

The musical model

Here, we define our musical model as a series of notes, each having three properties:

1. *Inter-onset interval (IOI)* - The time elapsed between the previous and current note onsets. For the first note, it is the time between the beginning of the piece and the note's onset. The unit of measurement for this property can be set globally; in this case, it is a 16th note value using a tempo of 120 BPM.

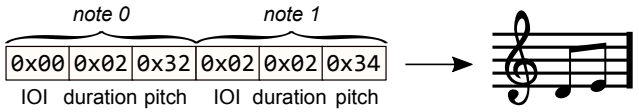

Figure 4: The model building process: parsing bytes to obtain note properties. The byte stream output from the virtual machine is sliced into three-byte chunks representing notes using one byte per property.

2. *Duration* - The current note's duration, in the same units as the IOI.
3. *Pitch* - A numeric value representing the note's pitch between 0 and 127, mapped to the same pitches as defined in the MIDI[1] protocol. The value 69 is associated with the 440Hz concert A, an increase or decrease of one unit representing a pitch rise or fall of one semitone respectively.

To build an instance of this model, the model builder parses the output of the virtual machine (see Figure 4). It masks each byte to give the boundaries of the three properties: Inter-onset interval and duration use 4 bits (longest note is the whole note), and pitch uses 7 bits (values are between 0 and 127). The model builder ignores any note whose duration or pitch is 0.

The resulting model is essentially a 3-row matrix, where each row represents a property, and each column represents a note. Given a model m, the matrix element $m_{2,n}$ is the pitch of the nth note. The model builder populates this matrix column-by-column.

Corpus-based fitness tests

The fitness of any rendered phenotype is determined by a series of similarity tests. All tests aim to evaluate how statistically similar a model is to the models in the corpus. This allows a large space of phenotypes to achieve high scores on the tests, not just the ones identical to a member of the corpus.

Here we take a holistic approach to building these tests, i.e. we test fundamental properties of our models instead

[1] Musical Instrument Digital Interface

of properties specific to music theory. We wish to find out whether looking only at the fundamental distributions' similarities allows the system to converge towards musical traits without the need to specifically test for them. This approach allows us to keep the generic aspect of this algorithm, so non-musical models may also benefit from it.

We define two types of tests: those revolving around a single property, such as total duration or overall number of notes, and those revolving around statistical property distributions.

We will refer to the overall number of tests as T. Given a model m and test index t, the test $f_t(m)$ returns a *grade* $g_{t,m}$, which is a numerical value between 0 and 1. The model's overall grade $\bar{g}_m$ is determined by a linear combination of the individual grades per test. We will refer to the coefficients of this equation as *importances*, since a higher coefficient means the test impacts the overall grade more. Every test f_t is assigned a predefined importance i_t. The importances are scaled to sum to 1 over all tests:

$$\sum_{t=0}^{T-1} i_t = 1, \tag{1}$$

therefore the overall grades will also be between 0 and 1:

$$g_{t,m} \in [0,1] \quad \Rightarrow \quad \bar{g}_m = \sum_{t=0}^{T-1} i_t g_{m,t} \in [0,1]\,. \tag{2}$$

Chosen corpus

We have chosen Bach's *Inventions and Sinfonias*, comprising 30 keyboard exercises, as our corpus. This catalogue of musical pieces was chosen for the following reasons:

- *Relatively short pieces of equal length* - For this research, we wish to test if, given a corpus of small length pieces, our system will converge to create pieces of similar length. Using this constrained length can help demonstrate that our algorithm finds the appropriate solution subspace where pieces of these lengths live.
- *Constant tempo* - We have not included tempo as a property which changes within the models (it is preset to 120 BPM) and we do not test for it.
- *Similar style* - Stylistically, all pieces of the corpus are similar in phrasing and property distribution. By using them, we once again limit our solution subspace and test whether or not the algorithm can find it.

The corpus is has been extracted from a set of MIDI files[2]. Our musical model has a structure similar to MIDI, therefore the algorithm has been extended to support conversion between the two representations.

The corpus files all comprise a single track of polyphonic music, therefore our system currently also produces single-track polyphonic music. The files in the corpus have been generated from a score rather then recorded by a human player. As a result they contain no *musical expression* such as changes in dynamics and tempo (e.g. all notes have velocity 127). For this reason our current systme does not use velocity as a property of notes within the models but it could easily be added at a later stage.

[2]Downloaded from `www.midiworld.com/bach.htm`

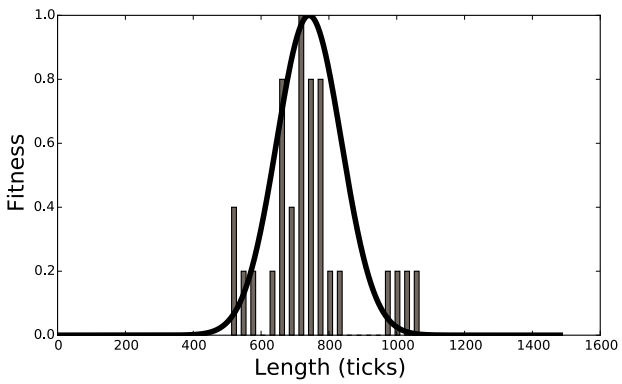

Figure 5: Deduced normal distribution test for musical model length. The length distribution for the pieces in the corpus (histogram bars) determines the grading schema we use for input models. The thick overlayed line shows the deduced normal curve.

Normal distribution tests

In our first corpus-based tests, we evaluate two single-value properties of a model: the *total duration* and *overall number of notes*.

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 are 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 5 shows the deduced normal of the length test and the histogram of the lengths of the corpus, to show the correlation.

Descriptor correlation tests

In this system, a descriptor is the output of a series of transform methods applied to an input model. The rest of our tests look at similarity between the descriptors of the input and that of the corpus. Four transforms are applied to each of the three properties of a model, resulting in a total of twelve correlation tests. The input for these transforms is always one property of all notes, i.e. it is a row of the model matrix.

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

Csaba Sulyok, Andrew McPherson, Christopher Harte (2015) Corpus-taught Evolutionary Music Composition. Proceedings of the European Conference on Artificial Life 2015, pp. 587-594

descriptors have a similar histogram for a property, it means the distribution of that property is similar. The associated test discourages unlikely values (e.g. very low or very high pitches), where the likelihood of a value is determined by the corpus.

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

Fourier transform The previous transforms lose all 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. Similarity of these properties may encourage the emergence of rhythmic patterns and meter.

Fourier transform of differential This transform shows the repeating patterns in the rate of change between consecutive values.

All these transforms produce an output of size 128, independent of the size of the input. Therefore a descriptor is encoded as a matrix of size 12×128.

We calculate the correlation between two descriptors line-by-line, because we want to be able to assign different importances to the transforms. Therefore we produce 12 grades.

To calculate the correlation between the same line of two descriptors, we use the *Pearson correlation coefficient* (r). Given the ith lines from descriptors D and E, denoted by D_i and E_i, and their mean values denoted by $\bar{D}_i$ and $\bar{E}_i$, their correlation coefficient is given by the following equation:

$$r_{D_i,E_i} = \frac{\sum_{i=0}^{127}(D_i - \bar{D}_i)(E_i - \bar{E}_i)}{\sqrt{\sum_{i=0}^{127}(D_i - \bar{D}_i)^2}\sqrt{\sum_{i=0}^{127}(E_i - \bar{E}_i)^2}} \quad (3)$$

The resulting coefficient is between -1 and 1. Positive values imply positive correlation, negative values imply negative correlation and a value of 0 implies no correlation. In our tests, we return 0 for negative values because we do not wish to obtain negative fitness scores. This approach allows us to return grades between the values of 0 and 1 without having to normalize the descriptors themselves.

Corpus clustering

We have defined correlation between two descriptors but our corpus consists of 30 different musical models, all of which have a different descriptor. Therefore a method must be used to incorporate the data from all 30 corpus members into the test. To verify how well this incorporation works, we evaluate the members of the corpus themselves using our tests.

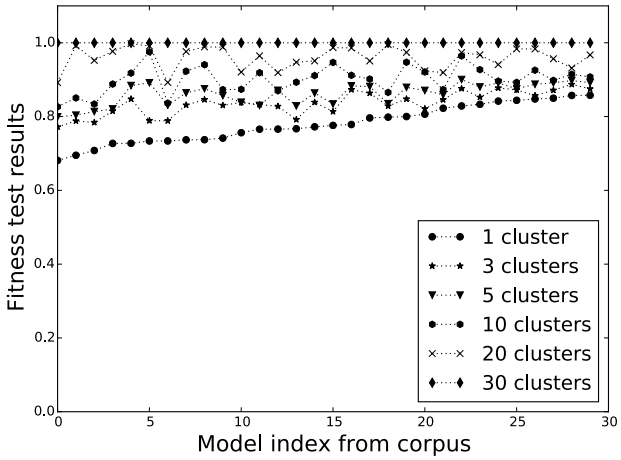

Figure 6: Similarity test results of corpus for different numbers of clusters; the models have been sorted by correlation.

A possible reference descriptor could be the centre of all descriptors from the corpus. This would be a *single niche* (Mahfoud, 1995) within our solution space. Experimenting with this approach reveals that testing the corpus itself gives relatively small grades (between 70 and 80%). This suggests that taking just one niche narrows down our solution space so much that even the corpus cannot achieve a high enough grade.

Another possible 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, this can become computationally expensive, and might broaden the solution space too much.

To obtain the best of both approaches, 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 a centre.

The clustering happens offline, before the evolutionary algorithm starts. It only needs to be done once therefore we do not have to worry about the computational expense. We use the *iterative partitional clustering algorithm* (Jain and Dubes, 1988), using the following steps:

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

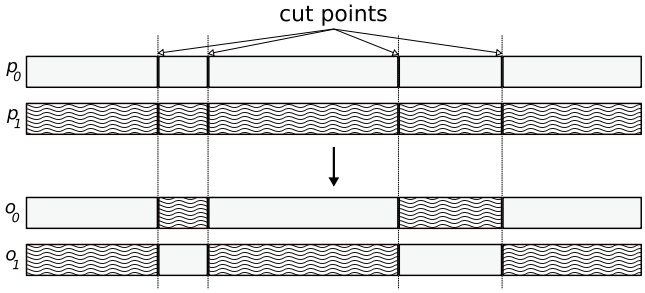

Figure 7: Visualization of crossover process; random cut points are chosen on the parents and offspring created from their recombination.

of K between runs. The two methods proposed initially can also be achieved by setting $K = 1$ or $K = 30$, respectively. Figure 6 shows the achieved grades of the corpus using different values for K.

Evolution strategy

Once all model grades have been calculated, we choose pairs of individuals for crossover. We use a *complementary phenotype selection* process (Dolin et al., 2002), which looks at individual test grades as well as overall scores. It chooses a mother based on roulette-wheel selection and builds hypothetical best-case offspring with all potential fathers. The hypothetical best-case offspring have the maximum grade for each test from the mother and father. The father is then chosen based on roulette-wheel selection applied to these grades.

When creating and breeding genetic strings, we do not concern ourselves with how this genetic string will be used later on, we simply view it as a byte array. In the first run of the algorithm, the generation-zero genetic strings are randomly generated byte arrays of the given size. In all subsequent generations, offspring are spliced versions of their parents. We will refer to the genetic strings of the parents as p_0 and p_1.

With the parents p_0 and p_1 chosen, the *genetic string breeder* builds two new genetic strings o_0 and o_1. It chooses a random number of cut points, separates p_0 and p_1 into chunks and populates the offspring (Figure 7). One chunk is taken from one parent, the next from the other.

Children o_0 and o_1 are then mutated by taking a random number of byte indices, and randomizing those bytes. The ratios of maximum cut points and maximum mutated bytes to the genetic string size are predefined for each run of the experiment.

Survival of the fittest is applied before the crossover process. A subset of the population can survive into the next generation. The ratio of the surviving units in a population is a predefined value. The units with maximum overall grades are allowed to survive into the next generation, but they are given an *age* value, so no phenotype can survive more than a given number of generations.

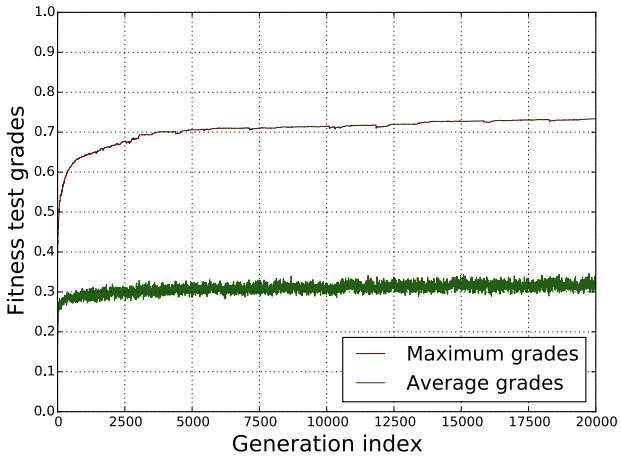

Figure 8: Progression of grades over 20000 generations: the mean and maximum grades for each generation, averaged over the 40 runs. The maximum grades steadily rise, while the mean grades seem to plateau quickly.

Experiments & results

For this paper we have run a total of 40 experiments, each time allowing the algorithm to reach 20000 generations; the parameters used here have been derived empirically from earlier test runs. We used a population size of 256 with a survival rate of 3% and number of clusters $K = 5$. The virtual machine's stopping conditions are a maximum of 60000 commands or 2500 output bytes. The breeder uses a 0.1% maximum cut point ratio and a 2% maximum mutation ratio.

In early experiments, it was found that some fitness tests tend to block others, preventing them from achieving high scores; importances have therefore been set to mitigate against this. For example, models with fewer notes have a higher probability of scoring well on correlation tests and this would block the emergence of desired lengths or number of notes. Therefore, the single property normal tests havea high importance.

The algorithm saves a number of files every 2500 generations. The first is the algorithm's current state, which can be imported for analysis or continuation. This file contains only the last generation. The second cyclically saved file holds all the test results since the past save. This way, none of them are lost, but only a limited number are in memory at any given time. The third file is a MIDI export of the highest-scoring model in the current generation.

To analyse the results of our series of tests, we take a look at the overall progression of the test results. Figure 8 shows the mean and maximum grades per generation, averaged over the 40 runs. We can observe a steady rise in

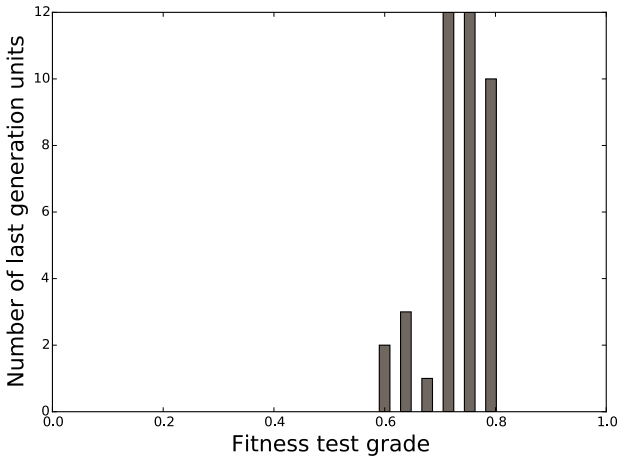

Figure 9: Distribution of highest-scoring individual grades. Every run managed to converge to at least a 60% correlation.

the maximum score, but stagnation in the mean values. This can be explained by the fragility of a genetic string when faced with crossover and mutation; even a small change can produce a completely different musical model.

Figure 9 shows the grade distribution of the highest-scoring individual in each run. It demonstrates that the algorithm gives consistent results on different iterations.

Examining the resulting MIDI files[3] allows us to evaluate, albeit subjectively, the musicality of the results. Their durations are universally within the boundaries dictated by the corpus statistics, although many achieve this with a smaller than desired number of notes (i.e. the notes are on average longer). Most exhibit recurring patterns of varying lengths, while some sound more random.

Figure 10 presents a selection from the results, demonstrating emergence of some musical properties, namely *repetition* and *variation*. Another property that emerges in rare cases is *segmentation*, i.e. longer sequences recurring at different times. Our tests did not specifically focus any of these properties, rather, they have emerged solely based on the similarity to the corpus.

Although repetition and variation appear in these pieces, other musical properties such as *harmony*, *melody* or *entropy*, are somewhat lacking. This suggests the need for further fitness tests inspired by music-theory.

Conclusion & future work

In this paper we have demonstrated a novel approach to evolutionary music composition: evolving the composition process rather than the product. By using a Turing-complete virtual machine, we have successfully defined our genotype as that process. Using a corpus of real music, we have derived tests to measure similarity of musical pieces to a set of niches.

Our results are promising, demonstrating that while this approach succeeds at converging towards the properties of the pieces in the corpus, is still only produces partially musical results. Some traits such as repetition and segmentation do arise, but there is still much room for improvement.

As a first step towards the reinforcement of the current ideas, we propose further test runs using different settings. The choice of parameters for the current run was largely empirical, therefore it may be interesting to assess differences arising from altering parameters such as population sizes, numbers of generations and survival rates.

For further research, we propose the inclusion of music-theory-inspired tests, which could accurately assess specific musical traits that do not yet emerge in the current context, such as harmony and entropy.

Improvements could also be made by a different choice of corpus. Since the current one had no musical expression, it would be interesting to explore what musical pieces played by human musicians could teach our system to do. Incorporation of velocity into the model properties would then allow us to test for accentuation and expressive tempo changes. We could also extend our corpus to include multivoice pieces such as string quartets. By testing the voices separately, the system could potentially detect the need for different pitch ranges or inter-onset intervals in the individual voices.

[3] All MIDI files and scores are uploaded to `http://csabasulyok.bitbucket.org/emc`

References

Biles, J. (1994). GenJam: A genetic algorithm for generating jazz solos. pages 131–137.

Biles, J. A. (2007). Improvizing with genetic algorithms: GenJam. In *Evolutionary Computer Music*. Springer London.

De Prisco, R., Zaccagnino, G., and Zaccagnino, R. (2011). A multi-objective differential evolution algorithm for 4-voice compositions. In *Differential Evolution (SDE), 2011 IEEE Symposium on*, pages 1–8.

Dolin, B., Arenas, M. G., and Guervós, J. J. M. (2002). 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, pages 142–152, London, UK, UK. Springer-Verlag.

Dostál, M. (2012). Musically meaningful fitness and mutation for autonomous evolution of rhythm accompaniment. *Soft Computing*, 16(12):2009–2026.

Eigenfeldt, A. (2009). The evolution of evolutionary software: Intelligent rhythm generation in kinetic engine. In *EvoWorkshops*, volume 5484, pages 498–507. Springer.

Eigenfeldt, A. (2012). Focus corpus-based recombinant composition using a genetic algorithm.

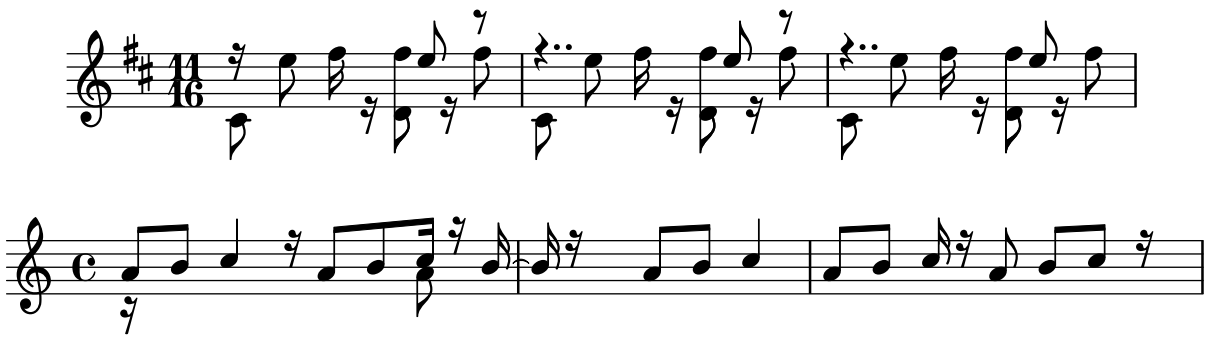

Figure 10: Two example segments from MIDI files created by the composition system. The recurring pattern in the top stave demonstrates that the algorithm helps repetitions emerge naturally. The lower stave shows a small three-note pattern with added decorations, demonstrating the evolution of variations. Time and key signatures have been added manually for illustration.

Fonseca, C. M. and Fleming, P. J. (1993). Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization.

Gibson, P. and Byrne, J. (1991). Neurogen, musical composition using genetic algorithms and cooperating neural networks. In *Artificial Neural Networks, 1991., Second International Conference on*, pages 309–313.

Hartmann, P. (1990). Natural selection of musical identities. In *International Computer Music Conference*.

Horner, A. and Goldberg, D. E. (1991). Genetic algorithms and computer-assisted music composition. In Belew, R. K. and Booker, L. B., editors, *ICGA*, pages 437–441. Morgan Kaufmann.

Jacob, B. (1995). Composing with genetic algorithms. In *International Computer Music Association*, pages 452–455.

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

MacCallum, R. M., Mauch, M., Burt, A., and Leroi, A. M. (2012). Evolution of music by public choice. *Proceedings of the National Academy of Sciences*, 109(30):12081–12086.

Mahfoud, S. W. (1995). Niching methods for genetic algorithms. Technical report.

Manaris, B. Z., Roos, P., Machado, P., Krehbiel, D., Pellicoro, L., and Romero, J. (2007). A corpus-based hybrid approach to music analysis and composition. In *AAAI*, pages 839–845. AAAI Press.

Martins, J. M. and Miranda, E. R. (2007). Emergent rhythmic phrases in an a-life environment.

McIntyre, R. (1994). 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*, pages 852–857 vol.2.

Miranda, E. R. and Biles, J. A. (2007). *Evolutionary Computer Music*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Phon-amnuaisuk, S., Tuson, A., and Wiggins, G. (1999). Evolving musical harmonisation. In *In ICANNCA*.

Rodriguez, J. D. F. and Vico, F. J. (2014). AI methods in algorithmic composition: A comprehensive survey. *CoRR*, abs/1402.0585.

Tokui, N. and Iba, H. (2000). Music composition with interactive evolutionary computation. In *Proceedings of the 3rd international conference on generative art*, volume 17, pages 215–226.

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

Unemi, T. (2002). SBEAT3: A tool for multi-part music composition by simulated breeding.

Waschka, R. I. (2007). Composing with genetic algorithms: GenDash. In *Evolutionary Computer Music*. Springer London.

Whorley, R. P., Rhodes, C., Wiggins, G., and Pearce, M. T. (2013). Harmonising melodies: Why do we add the bass line first? In *International Conference on Computational Creativity*, pages 79–86, Sydney.