# Music Composition by Interactive Evolutionary and Genetic Computation

Neil Babson

CS 541

This program uses two slightly different artificial intelligence algorithms to create original musical compositions that are each eight measures in length. The first algorithm creates a population of successful single measure musical elements by a process of evolution whereby the most fit individuals from a given generation each produce a single mutant offspring. After a group of fit measures has been developed the second algorithm uses genetic mating to combine the features of the best eight-measure musical specimens until a single composition is produced. Both of these processes are 'interactive', meaning that the fitness measure is provided by the human user who assigns a value to each individual according to their subjective pleasantness, or by some other criteria. The program was written in Haskell, using the Haskore music library, which writes objects of the Music data type to MIDI files.

The evolutionary algorithm starts by randomly generating a population of sixteen measures, with the notes drawn from two measures of the C Major scale. Random selections are made using an infinite list of random numbers, which are discarded from the front of the list as they are used. Infinite lists are possible because of Haskell's policy of lazy evaluation, according to which no calculations are actually performed until their result is required. Each tone has a duration, ranging from sixteenth note to whole note; with the longest durations, dotted half note and whole note, selected half as often as the rest. Each note also has a starting time ranging from the beginning of its host measure to 15/16$^{th}$ of the way through. Tones can therefore extend past the end of their respective whole-note-length measures,

sustaining into the next measure when they are juxtaposed to create longer musical pieces. Each tone is represented by a (note, start time, duration) 3-tuple, and each measure by a list of 3-tuples. The lists can then be easily converted into Music objects and played as MIDI files.

In each generation of the algorithm the user assigns each measure a fitness value ranging from zero to nine. The fitness criteria can be adjusted to steer the musical development towards a particular style, or can be a pure reflection of the user's aesthetic response. The measures are then sorted by fitness and the bottom half are discarded. Each of the survivors creates a new offspring by mutation, and the next generation, consisting of the survivors and the mutant offspring, is shuffled before being presented again to the user for ranking. A mutation consists of ten random changes to an individual measure. Three kinds of change can occur: dropping a note (nothing happens if the measure already has no notes), adding a note, or changing a note, the last being a composition of dropping and adding. The measures will continue to evolve until half (or more) of them get assigned the highest fitness rank of nine.

I experimented a little bit with this algorithm during the program's development. At first it seemed reasonable to make the amount of mutation that a survivor's offspring underwent depend (inversely) on how well the survivor was ranked. This approach failed by producing a cluster of very similar good measures, lacking the diversity for interesting music. I briefly tried dealing with this problem by discarding the survivors, so that only mutants reached the next generation. This, unsurprisingly, led to a sort of random walk, in which the quality of the overall population didn't noticeably improve. These attempts led me to conclude that all surviving individuals must be mutated by the same amount. Ten changes per mutation, after some trial and error, was found to generally preserve a recognizable relationship between parent and offspring will also providing enough novelty for evolution to proceed at a reasonable rate.

With the completion of the evolutionary algorithm a set of eight eight-measure 'songs' are generated

from the eight measures that passed the previous stage. The random selection seemed preferable to a more equitable distribution of the surviving measures, despite the small possibility that they are not all used, because it creates a more diverse initial song population. As in the previous algorithm the population is sorted by user ranking and the bottom half is dropped. The next generation is composed of the four offspring and four children which are produced by mating the first and fourth best survivors as well as the second and third best ranked.

Mating is accomplished by breaking each parent at the same internal inter-measure division and reattaching the head of each parent to the tail of the other. This is easy to accomplish because the songs are stored as eight-tuples of measures. It quickly became apparent that this algorithm could be improved by the introduction of a mutation that changed which measures were at the beginning and end of songs. Accordingly, each child has a one in three chance of being shifted between zero and seven measures, while otherwise preserving their order. Genetic evolution continues until one or more songs get a ranking of nine, at which point one of these successful songs is selected and saved as a MIDI file before the program terminates.

The main challenges encountered in the course of this project were due to my limited familiarity with Haskell. Some minor difficulties were occasioned by the lack of familiar loop structures, as I applied familiar programming techniques in a purely functional, rather than an imperative environment. More challenging were my first encounters with the IO Monad and the fact that values, such as an Int, which have been sullied with exposure to IO can never be changed back into a pure Int. I tried many ways to make this conversion before realizing that it was impossible. Eventually I did learn how to use 'do' blocks to deal with the Monad values, and I'm confidant that the program is bug free and works as intended.

The code consists of a mere 273 lines, of which at least a third is either comments or empty lines; but thanks to the terseness of Haskell, this was enough to fully implement two (very similar) AI

algorithms. All of the code was written by myself with the exception of a seventeen line shuffle algorithm at the top of the composer.hs file (after the imports) which was taken from wiki.haskell.org/Random_shuffle. In addition, the program makes use of an imported module, 'Music', which was provided by professor Tim Sheard, and contains the helper functions 'writeMidi' and 'playWin'.

An obvious drawback to this sort of interactive learning algorithm is that it requires a human user to spend a lot of time steering the system towards a desirable outcome. If one is willing to invest the effort, this program is capable of producing musical works of surprising beauty. I'll include in the project .zip file my favorite example, named 'finishedSong3', which required 41 evolutionary generations and 16 genetic generations to complete.