Genetic Algorithm for Composition of Musical Melody Lines

Project Overview

By: Ari Brown

11/11/16

Purpose:

The structures and syntax of Western music theory have been described as resembling nature in many ways, and many theorists have looked at the compositions of Mozart and Bach from an algorithmic perspective. For example, Bach's work has been known to most closely match with fractal-based patterning, which expresses itself through nature in forms such as seashells, ferns, shorelines, and snowflakes. So why not use an algorithmic process that is "natural" to produce more of this style of music?

This project will use genetic algorithm in order to optimize the most interesting and best sounding melody lines. A genetic algorithm will hopefully produce melodies that are not completely formulaic (otherwise, why not just create perfectly structured music using recursion), but are based on a defined set of rules that mold them into better forms.

Music Theory Overview:

Melody lines can be analyzed in a variety of ways, and here are the things I will try to include in this project:

- Analysis of diatonic fit- in other words, how much of the notes in the melody fit within a set key?
- Chord outline- melodies often follow chord progressions, in which they arpeggiate notes in succession that suggests a certain chord. There are a variety of different chords associated with a given key, and better melodies usually fall within this patterning
- Progression outline- the order of combination of chords in a melody matters because different chords have different functions. Some are used more often during climactic areas, some have a lot less tension, and some are extremely stable harmonically
- Rhythmic distribution- Notes that outline chords or match key are usually found on the strong beats in measures, and so they are emphasized more (e.g. beat 1 in 4/4)
- Self similarity- melodies can be self similar in terms of the intervals (distance between two adjacent notes) present in the music, and in terms of the rhythmic layout of those notes. Self similarity falls in line with repetition of musical motifs, which is important to musical expression

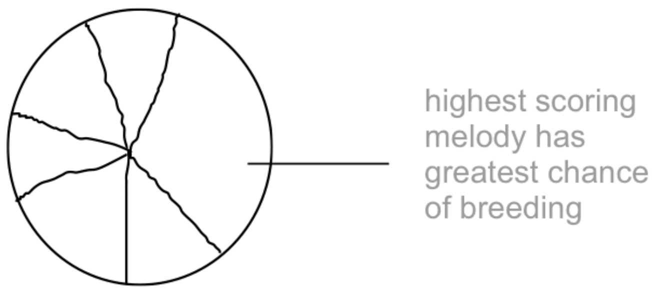
Genetic Algorithm:

The genetic algorithm will work to "breed" the best sounding melodies through a simulation of generations of evolution, in which natural selection takes place and the better "more fit" melodies are more likely to survive and reproduce. The algorithm will work as follows:

- Start out with an initial population of melodies of a given size
- Assign a score to each melody based on the music theory traits
- Choose two melodies at random from a probability distribution that is roulette wheel style (melodies that have a higher score are more likely to be picked)
- Breed those two melodies, creating a new child melody, which will include half of the traits from one parent melody and half from the other, mutations are also incorporated here
- Repeat the process of breeding between two parents until the next generation target size has been filled
- Repeat all of the steps with the newly created generations, until stop conditions have been met

*Note about stop conditions: the stop condition can be user-specified, where a certain amount of generations to run through is predetermined, or the genetic algorithm could incrementally decrease generation size until there is only one child in the last generation

*Roulette Wheel Diagram:



Scoring Function:

The scoring function will determine the fitness of each melody and assign an arbitrary score value to the melody. The scoring function will take into account all of the traits mentioned in the music theory overview, and different methods will be needed to obtain sub-scores based on each trait. The scoring function will add up the results of the following:

Diatonic Fit	The score will be added to when notes in the melody match notes in the desired key. These points will be weighted by where the notes fall rhythmically as well. Notes that are on strong beats that are in the key will score more points that notes on weak beats in the key. Notes not in the key won't earn points
Chord Outline	The scoring method for chord identification is a little tricky. This algorithm will start at a note, and consider it a one-note chord (if the note matches the key, diatonic fit can be accounted for here). Then, it will consider the adjacent note with the current note as a two-note chord. Points will be added to the score each time there is a match to a chord when the next adjacent note is added. There will also be an acceptance of failure of at least one note, meaning that the melody can contain a note that is off and then continue to stick to the chord. This process will be repeated for all notes
Progression Outline	Looking for chord progressions in the melody can be done while identifying the chord outline (in that algorithm). Once a chord has been identified, it can be marked as tonic, predominant, or dominant in function. The most favorable progressions are of the order tonic, predominant, dominant, and so that specific order should receive a high score. Dominant to tonic is also common and will receive points
Self Similarity	The Jaccard distance between two sets of notes can be measured recursively. At the first level, the distance will be measured from the left half to the right half of the melody, and from there on, the distance will be measured between sub-halves. Smaller distances between two sets of notes indicate more repetition and self-similarity, and should therefore be scored higher than bigger distances
Range/Interval Smoothing	Smoother melody lines will get higher scores. Smoothing can be accessed based on the scale degree relative to the key (1-7, or 1-12 chromatically). Smoothing can also be scored based on range, where smaller jumps are better (we don't want a melody jumping randomly up and down all the time, which is likely in a random set of notes)

Mutations:

Mutations can happen at different levels: note specific mutations and melody specific mutations.

Note Specific	Melody Specific
• Raise by half step • Lower by half step • Raise by whole step • Lower by whole step • Raise by octave • Lower by octave	• Rhythmic redistribution- redistribute rhythm within beats between two adjacent notes • Thematic generation- copy the rhythm or sequence of intervals somewhere in the melody, and impose it on a different part • Smooth melody- place notes so they are closer together in register

*Note: certain types of mutations can be more likely to happen in different stages of the genetic algorithm; the chances can incrementally change at each iteration. For example, the chance of the the rhythmic redistribution mutation may be large in the first generations and smaller later, in order to vary rhythm at first, but then solidify thematic material

---

Data Structures:

The main data structure to be used in storing the melodies is a tree-style structure. In the case of the genetic algorithm, the tree will start off with many parent nodes (if the population size keeps decreasing, the tree will take on the form of an inverted tree). Any new child melody created will be stored in a node pointed to by the parents.

A node will have the following elements:

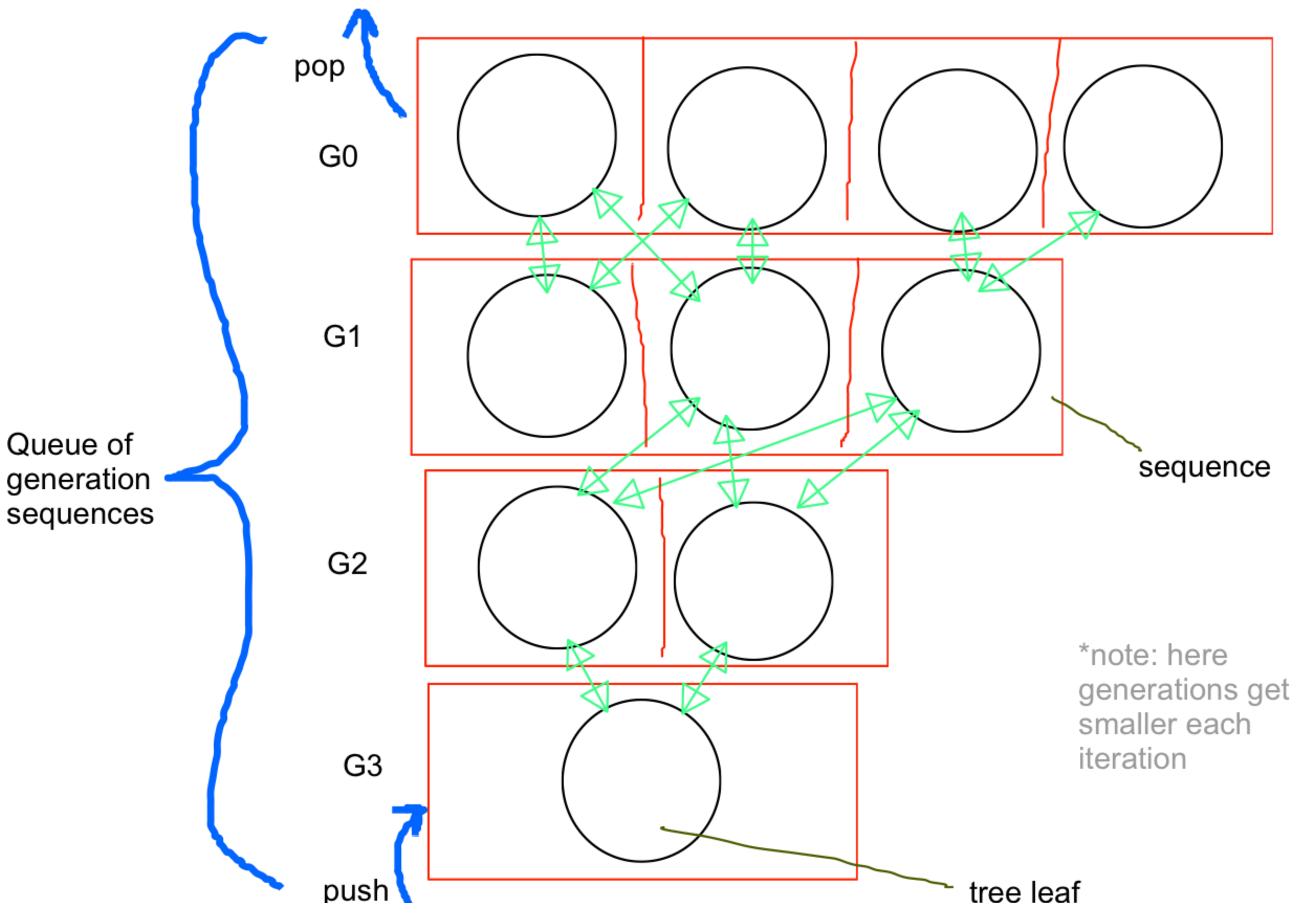
- A melody, the main data in each node
- A score associated with the melody
- A vector storing pointers to all of that node's children
- Two pointers for the node's parents (mom and dad), which makes the tree doubly-linked

Storing the melodies in a tree form will allow for the user to not only experience the program's output of the most evolved melody, but also look back in that melody's ancestry. The way of storing the melodies would allow for easy access to the family tree, and there could also be a function that allows the user to search for any cousins of a particular melody.

In addition to storing the node pointers in a tree, it is also useful to store the same pointers in sequences- one sequence for each generation. This is necessary for the way the algorithm progresses. The GA will start off with a sequence of parent nodes, and the pointer to a new child will be linked with the parents, as well as put into a sequence of nodes in that generation. This way, recursively accessing the leafs (current-most generation) does not have to be done as each generation progresses, which changes the time complexity greatly (from n*generations to constant time). In addition, it would be very difficult to set up the roulette-style selection for breeding without an array of the current generation's melodies.

In addition, to save memory space, the user will be able to specify how many generations they would like to have saved. The sequences of melodies that make up each generation can then be saved in a queue, which deals with the memory in a first-in first-out fashion. For example, if the user specifies 7 generations be stored, one the queue (which holds generation sequences) fills to 7, whenever a new generation is added, a generation is popped off the queue and destroyed.

Diagrams of the structures:


pop
G0
G1
G2
G3
push
Queue of generation sequences
sequence
tree leaf
*note: here generations get smaller each iteration

Stretch goals:

- Use a sorting algorithm to sort the last generation by score
- Integrate a GUI into the program that allows users to set genetic algorithm parameters, and allows them to see the melody results in a score form.
- Use LilyPond Library to generate the notated music to these melodies.
- Use the PortAudio C++ library to create MIDI files of the resulting melodies, and generate audio playback of the music
- Write an algorithm that generates a bass line counterpoint based on the melody generated, or generates four-part harmony based upon the melody
- Extend the genetic algorithm to a full piece of music, taking into account sections and modulation to different keys

Schedule:

Week 1	Implement all of the basic abstractions (note, melody, generation classes) along with their test files. Also implement the tree data structure, the sequence data structure, and the queue structure (a queue of sequences), with test files
Week 2	Implement all scoring functions which lays the foundation for the genetic algorithm, add any other mutation functions to the note or melody classes
Week 3	Piece together all of the implementations in the genetic algorithm, write main so that user can input GA parameters