A Genetic Algorithm Approach to Improve Automated Music Composition

Nathan Fortier, Michele Van Dyne

***Abstract*—** Using the rules of music theory, a program was written which automatically creates original compositions. These compositions were parameterized by user input concerning preferences on genre, tempo, and tonality. Based on these preferences, initial compositions were generated, and the "best" composition was presented to the user. Following the rules of music theory guarantees that the program produces harmonious compositions, but certain aspects of musical composition cannot be defined by music theory. It is in these aspects of musical composition where the human mind uses creativity. Using the population of compositions initially generated for the user, the program then used a genetic algorithm to evolve compositions that increasingly match the user's preferences, allowing the program to make decisions that cannot be made using music theory alone. The resulting "best" composition of the evolved population was then presented to the user for evaluation. To test the effectiveness of this approach, each composition, both initial and final was ranked by subjects on a scale from 1 to 10. Subjects expressed a significant preference for the evolved compositions over initial compositions.

***Keywords*—** Artificial creativity, artificial intelligence, genetic algorithms, machine learning, music

I. Introduction

THIS research describes a program using rules of music theory to create original compositions and then using a genetic algorithm to utilize the preferences of listeners and generate qualitatively better compositions. Following the rules of music theory guarantees harmonious compositions, but certain aspects of musical composition cannot be defined by music theory. It is in these aspects of musical composition where the human mind uses creativity. To test the effectiveness of the approach, subjects were interviewed about certain musical preferences, and these preferences were used as a basis for the fitness evaluation function of the genetic algorithm. The results of the research were evaluated both quantitatively, based on the program's own fitness evaluation of the compositions, and qualitatively, based on subject ratings of compositions. In this paper, we first discuss current research in automated music composition, then the basis for genetic algorithms, and finally basics of music theory. We then describe the operation of the algorithm, experimental design and results, and conclusions and future directions based on those results.

Manuscript submitted August14, 2011.

N. Fortier is master's student in Computer Science at Montana State University, Bozeman, MT, USA. He graduated with his B.S. in Software Engineering from Montana Tech, Butte, MT 59701 USA. (phone: 406-498-1303, e-mail: nathan.fortier@gmail.com).

M. Van Dyne, PhD, is an associate professor of Computer Science at Montana Tech, Butte, MT 59701 USA. (phone: 406-496-4855, e-mail: mvandyne@mtech.edu).

A. Current Research in the Literature

The study of artificial creativity in the domain of music has focused on both compositional and improvisational music. Several projects available on the web show the innovation of such studies. Work in the domain of compositional music has been done by David Cope with his Experiments in Musical Intelligence [1], [2], a computer system that deconstructs existing music into separate parts and recombines those parts to from new musical works. Work in this domain has also been done by the developers of Wolfram Tones [3], a software system that takes an arbitrary program from the "computational universe" and uses that program's output to compose music. Work in the domain of improvisational musical creativity has also been done at Georgia Tech [4] where researchers have developed a robot that can play improvisational jazz on a marimba.

Beyond the public projects, more focused research on musical composition can be found in the literature. Klinger and Rudolph [5] have used an evolutionary approach to simulate creativity in generating musical melody lines. They then evaluated the melodies in two automated methods: an artificial neural network, and a decision tree. These evaluations were compared to human (interactive) evaluations of the same melodies. They found that the decision tree produced by the C4.5 data mining algorithm more closely agreed with human evaluators than did the artificial neural network. The significance is that if the evaluation can be automated, the bottleneck of human interaction can be reduced or eliminated. Our research also seeks to eliminate the bottleneck of human interaction, though in our work it is evaluation of an entire composition and not just the melody lines.

Alfonseca, Cebrian and Ortega [6] used the normalized compression distance as a fitness function to evaluate compositions generated from a given composition. As an example used in their paper, if we wish to generate compositions similar to those of Mozart, then the system would be seeded with one of (or several of) Mozart's compositions. In this approach, the intent was to choose a piece of music from a given artist and then allow the system to compose music similar to that style. In addition to the standard recombination and mutation operators used in evolutionary algorithms, they also used fusion and elision. Fusion allows the genotype to be concatenated with a piece broken either from itself or from a sibling and elision allows one gene to be eliminated. Different permutations of these operators were used in combination to produce different strategies, and each strategy was evaluated for effectiveness. It was found that a combination of strategies based on different points in the learning curve produced the best results, and that fusion doesn't seem to produce better results, but elision does. Our current research allows the user to select a genre rather than a composer, and the system attempts to compose music according to preferences rather than similarity to a particular artist.

Maddox and Otten [7] used an evolutionary algorithm to generate four-part 18th century harmony in the style of J.S. Bach. They stressed that randomly generating musical notes would be entirely too time consuming to be effective, and that it was necessary to have reasonable constraints in generating the music and to have a method for automating the quality of the results. Following from their results, our research uses music theory to constrain the generation of compositions and uses a fitness function for automated evaluation which is based on user-expressed preferences.

An overview of research on evolutionary computation in this domain is discussed in [8]. The authors categorize existing research according to four types: interactive, example-based, rule-based, and autonomous systems. Throughout the categorization, the roles of creator and critic, or author and audience, are used in relation to the systems. Interactive systems, in general, use automated composition as the creator and have a human listener as the critic. An issue with these systems is that the human critic is a bottleneck in evaluation. The second type of system, example-based, includes a user as the creator who provides examples to a system (usually an artificial neural network) which then learns specified themes from the presented examples. In a sense, then, the human is both the creator and critic initially, but the systems learn to become the critic over time. The third type of system is rule-based. In this type, a human provides rules to evaluate system generated compositions, again placing the human in the role of critic and the system in the role of creator, though this does not appear to have the same bottleneck effect as the interactive systems do. Finally, the last type of system is the autonomous one. In these systems, both critic and creator are automated; that is, the system creates its own compositions and then evaluates them. The authors comment that some of the drawbacks inherent in each type of system can be overcome by the development of hybrid systems which incorporate aspects of each type.

The research discussed in this paper incorporates aspects of three of the types of evolutionary systems discussed in [8]. The interactive aspect is addressed with human input of musical preferences and then evaluation of the best system generated compositions both initially and after evolution. In this aspect, the human is the critic and the system is the creator. The third type of system discussed was rule-based. Our research uses the rules of music theory to generate initial compositions. A human encoded these rules into the program, which form a first pass at composition evaluation, and the system is still the creator. Finally, our research also addresses the autonomous type of system in that it composes its own music, and in the absence of human intervention or evaluation, it evaluates its own compositions according to fitness measures; thus, it serves as both creator and critic.

Section two of this paper describes the architecture of our software and its approach to music composition and evaluation such that it follows the results of existing research but adds to these approaches.

B. Genetic Algorithm Background

A genetic algorithm (GA) is a search algorithm that imitates the process of natural evolution. Every genetic algorithm has a population containing a number of individuals, each of which contains some number of chromosomes. A chromosome is defined by an array of values that define some characteristic of the individual. A GA begins with an initially random population. Once the population has been initialized, selection, crossover, and mutation are used to create a new member of the population. During the selection process two members of the population are selected to be parents using fitness proportionate selection. Once two members have been selected the crossover process begins.

The crossover process contains two steps. First, for each chromosome the individuals contain, a single crossover point is selected. Second, all values beyond that point in both chromosome strings are swapped between the two parent individuals. This results in a child individual who then undergoes the mutation process.

During the mutation process some of the values in each of the individuals chromosomes are selected for mutation at random based on the GA's mutation rate. These values are then changed or mutated.

A more thorough explanation of genetic algorithms can be found in [9].

C. Music Theory Background

A piece of music is written by specifying a series of parts. Each of these parts is written by specifying a series of notes which are defined by their duration and pitch.

In western music the standard unit used to measure a note's duration is the whole note, whose length is typically equal to four beats. Most other note durations are described as fractions of the whole note. For example, a half note is half the duration of a whole note or two beats. The amount of time between each beat is determined by the tempo of the piece. Every musical piece can be broken up into measures which are defined by the song's time signature. A time signature consists of two numbers. The first number indicates how many notes are in each measure, while the second number defines the duration of each of these notes. For example, in the time signature 3/4 each measure contains three quarter notes, while in the time signature 6/8 each measure contains six eighth notes.

In western music nearly all pitches of a song are part of a scale or sequence of pitches. The two most common musical scales in the west are the major scale and the pentatonic scale. Most music revolves around some kind of chord progression or series of chords. A chord is a set if pitches are heard simultaneously. Most music contains three note chords called triads. The most common of the triads are the major and minor chords. Songs that begin and end on major chords are said to have major tonality while songs that begin and end on minor chords are said to have minor tonality.

An in depth explanation of music theory can be found in [10].

II. SYSTEM FUNCTIONALITY

Previous research in compositional and improvisational music has focused on the generation of music through different means, and certainly to emulate certain different styles. However, none have addressed the qualitative experience of human listeners, nor attempted to allow the music to evolve toward listener preferences. This research uses music theory principles to generate an initial composition, but then allows a user to specify preferences, and then learns or evolves toward that user's preferred style.

A. Program Description

The program begins by defining a desired style for the genetic algorithm to optimize toward and a population to store individual genetic codes and generate new genetic codes. After this the program enters a loop in which the following four step process in repeated:

- Step 1: Program calls the population's getCode() function to obtain a new genetic code.
- Step 2: A song generator, *G*, is created using the genetic code obtained in step one and a song is generated by *G*. During this process the generator determines the style of the genetic code.
- Step 3: The genetic code's style is then compared to the desired style and a value is obtained quantifying how close the genetic code's style is to the desired style. This value becomes the genetic code's fitness.
- Step 4: The new genetic code is inserted into the population

1. Obtaining the Genetic Code

Each song's genetic code consists of four chromosomes: an Arpeggio Chromosome that defines what notes of a chord are played when a part plays an arpeggio; a Melody Chromosome that determines the pitch and duration of each note in the melodies; a Rhythm Chromosome that determines the duration of notes in the song's rhythms; and a Basic Structure Chromosome that determines things like key, instrument voices, and chord progressions. The population's getCode() function begins by checking the population size. If the population is smaller than the initial population size then getCode() simply sets the above chromosomes equal to random values and returns the resulting genetic code. If the population is larger than the initial population size then getCode() performs the following steps:

- Step 1: Two members of the population are selected using fitness proportionate selection.
- Step 2: If both selected individuals have above average fitness then the function proceeds to step three otherwise the function returns to step one.
- Step 3: Crossover and mutation are performed on all chromosomes of both selected individuals.
- Step 4: A new code is created and returned using the resulting chromosomes.

2. Generating a Song and Defining a Style

The Generator begins creating a song by obtaining the song's tempo, voicing, time signature, and number of movements from its genetic code. It then generates each of the song's movements.

a. Generating a Movement

The Generator creates movements using the generateMovements() function, which returns a pattern that can be used as a movement in the song. This function begins by consulting the genetic code's Basic Structure Chromosome in order to determine the number of verses that will be in the movement. The function then generates each of these verses and adds each verse to a pattern that is then returned.

b. Generating a Verse

When a verse is created its constructor initializes a series of values such as the number of chords, tempo, time signature data, and voicing of the verse. It then calls the verse's generateVerse() function. This function begins by generating the verse's chord progression. The function then generates the verse's beat, rhythm, bass-line, harmonies, and melody in that order. If the verse is to contain more than one melody the harmonizing melody is generated last. Each of these parts are then added to the verse and the verse is transposed to the appropriate key.

i. Generating a Beat

When a beat is generated its constructor begins by initializing its time signature. The constructor then consults the genetic code's Rhythm Chromosome to determine the instrumentation of the percussion in the beat. Finally the beat's generateBeat() function is called. This function begins by generating the beat's ride part, low accent part, and high accent part in that order. Each of the parts are than assigned to an instrument based on the instrumentation chosen by the Rhythm Chromosome in the constructor. These parts are then added to a pattern and that pattern is returned.

ii. Generating a Rhythm

When a rhythm is created its constructor begins by initializing the rhythm's chord progression, number of measures, rhythm chromosome, voice, and time signature. The constructor then calls the rhythm's generateRhythm() function. This function first calculates the number of 16th notes in each measure based on the time signature. This value is assigned to the variable notesLeft. Next, it consults the Rhythm Chromosome to determine the duration of the first note in the rhythm. This value is assigned to the variable i. While notesLeft is greater than zero the following steps are repeated:

- Step 1: Set the variable dur16 equal to the number of 16th notes in i.
- Step 2: Set the variable duration equal to a character representation of i.
- Step 3: Set i equal to a new duration value obtained by consulting the Rhythm Chromosome.
- Step 4: If (notesLeft-dur16) is equal to zero goto step 1
- Step 5: Set notesLeft equal to (notesLeft-dur16)
- Step 6: Add duration to the rhythm

The generateRhythm function then loops the rhythm generated above for each measure. You will notice that the generateRhythm() function generates durations but not pitches. This is because the the rhythm's pitches are already defined by the verse's chord progression.

iii. Generating a Harmony

When a harmony is generated its constructor is passed several parameters, one of which is the parameter chordProg, which is an array containing each chord of the verse's chord progression. The constructor begins by creating the array pitches. Next, the constructor calculates the number of 16th notes in each measure based on the time signature. This value is assigned to the variable notesLeft. Another variable j is set to zero. The constructor then repeats the following steps for each note, r, in the rhythm:

- Step 1: Set *p* equal to a note in the chord denoted by *chordProg[j]*
- Step 2: Add *p* to *pitches*
- Step 3: Set *dur16* equal to the duration in 16th notes of *r*
- Step 4: *Set notesLeft* equal to *(notesLeft-dur16)*
- Step 5: If *notesLeft* is equal to zero increment *j* and set *notesLeft* equal to number of 16th notes in the measure

After this a harmony pattern is created in which each note's duration is that of the corresponding rhythm note's duration and each note's pitch is defined by the array *pitches*.

iv. Generating a Melody

When a melody is created its constructor begins by consulting the melody chromosome in order to determine what scale the melody is to use. After this the functions generateDurations() and generatePitches() are called.

The generateDurations() function determines the durations of all notes in the melody. It begins by obtaining a duration from the melody chromosome. This value is assigned to the variable *potentialDuration*. For each note in the rhythm that the melody must harmonize with, determine if the note the melody will play is longer or shorter than this note. If the note the melody will play is shorter than the note it is to harmonize with then do the following:

- Step 1: Set *notesLeft* equal to the number of 16th notes in the note to harmonize with.
- Step 2: Set *dur16* equal to the number of 16th notes in *potentialDuration and set the variable duration equal to potentialDuration*
- Step 3: Set *potentialDuration equal to a new duration obtained from the melody chromosome.*
- Step 4: If *(notesLeft-dur16)* is less than 0 then goto step 2
- Step 5: Set *notesLeft* equal to *(notesLeft-dur16)*
- Step 6: Add *duration* to the array that is to contain durations of all melody notes.
- Step 7: If *notesLeft* is greater than zero then goto step 2

If the note the melody will play is longer than the note it is to harmonize with then do the following:

- Step 1: Set *dur16* equal to the number of 16th notes in the current note to harmonize with
- Step 2: While *dur16* doesn't equal 8 or 4 and *dur16* is less than 16 set *dur16* equal to *dur16* plus the number of 16th notes in the next note to harmonize with
- Step 3: If *dur16* is less than or equal to 16 then set the string *duration* equal to the string equivalent of the duration described by *dur16* and add *duration* to the array that is to contain durations of all melody notes
- Step 4: If *dur16* is greater than 16 then undo all actions performed in the above steps and have the melody play a note that is shorter than the note it is to harmonize with

The generatePitches() function determines the pitches of all notes in the melody by doing the following for each duration in the melody:

- Step 1: If the duration is a whole note or half note then set the next pitch to an arpeggio of the chord it is to harmonize with.
- Step 2: Else consult the melody chromosome to determine if the next pitch will be a note of a scale or a note of an arpeggio of the the chord it is to harmonize with and set the next pitch accordingly.

v. Generating a Harmonizing Melody

When a harmonizing melody is created its constructor begins by consulting the melody chromosome in order to determine what scale the melody is to use. After this the functions generateDurations() and generatePitches() are called.

The generateDurations() function begins by obtaining a duration from the melody chromosome. This value is assigned to the variable *potentialDuration*. For each note in the first melody that the harmonizing melody must harmonize with, determine if the note the harmonizing melody will play is longer or shorter than this note. If the note the harmonizing melody will play is shorter than the note it is to harmonize with then do the following:

- Step 1: Set *notesLeft* equal to the number of 16th notes in the note to harmonize with.
- Step 2: Set *dur16* equal to the number of 16th notes in *potentialDuration and set the variable duration equal to potentialDuration*
- Step 3: Set *potentialDuration equal to a new duration obtained from the melody chromosome.*
- Step 4: If *(notesLeft-dur16)* is less than 0 then goto step 2
- Step 5: Set *notesLeft* equal to *(notesLeft-dur16)*
- Step 6: Add *duration* to the array that is to contain durations of all harmonizing melody notes.
- Step 7: If *notesLeft* is greater than zero then goto step 2

If the note the harmonizing melody will play is longer than the note it is to harmonize with then do the following:

- Step 1: Set *dur16* equal to the number of 16th notes in the current note to harmonize with
- Step 2: While *dur16* doesn't equal 8 or 4 and *dur16* is less than 16 set *dur16* equal to *dur16* plus the number of 16th notes in the next note to harmonize with
- Step 3: If *dur16* is less than or equal to 16 then set the string *duration* equal to the string equivalent of the duration described by *dur16* and add *duration* to the array that is to contain durations of all melody notes
- Step 4: If *dur16* is greater than 16 then undo all actions performed in the above steps and have the melody play a note that is shorter than the note it is to harmonize with

The generatePitches() function determines the pitches of all notes in the harmonizing melody by doing the following for each duration of the harmonizing melody:

- Step 1: If the duration is a whole note or half note or if the note the pitch must harmonize with is not part of an arpeggio then set the next pitch to an arpeggio of the chord it is to harmonize with.
- Step 2: Else consult the melody chromosome to determine if the next pitch will be a note of a scale or a note of an arpeggio of the the chord it is to harmonize with and set the next pitch accordingly.

c. Defining a Style

As the generator composes the song it defines the genetic code's style. A style is defined by the following ten evaluation criteria:

- Tempos Used (TM): an ArrayList of all tempos used in the style.
- Tonality (TN): a string describing tonality of the style as either major, minor, or ambiguous.
- Time Signatures Used (TS): an ArrayList of all time signatures used in the style.
- Pitch Variation (PV): the relative standard deviation of pitches in the melodies.
- Note Duration Variation (NDV): the relative standard deviation of note durations in melodies.
- Rhythmic Complexity (RC): the relative standard deviation of note durations in rhythm.
- Number of Parts (NI): the number of instruments used in the style.
- Instruments Used (TI): an ArrayList containing all instruments used in the style.
- Percent of Jazz Chords: (PJC) the percentage of triads that have added pitches.
- Scales Used (SU): an ArrayList containing all scales used in the style.

3. Determining a Genetic Code's Fitness

Every style in the system contains a compareTo() function that takes a style as a parameter and returns a value between zero and one. A values of one is returned if the two styles are identical and a value of zero is returned if the two styles have nothing in common. A code's fitness is determined by calling its style's compareTo() function, and the higher the returned value, the higher the fitness.

The compareTo() function calculates its total return value (RV) using each of the evaluation criteria as follows:

$$DIFF_X = \frac{\sum_i \sum_j (x_{desired_i} == x_{actual_j})}{\max(size(i), size(j))} \tag{1}$$

where X = the evaluation criteria of TM, TS, TI, and SU.

$$DIFF_X = 1 - |x_{actual} - x_{desired}| \tag{2}$$

where X = the evaluation criteria of PV, NDV, RC, and PJC.

$$DIFF_{TN} = ((TN)_{actual} == TN_{desired}) \tag{3}$$

$$DIFF_{NI} = 1 - \left| \frac{(NI_{actual} - NI_{desired})}{(NI_{actual} + NI_{desired}) \div 2} \right| \tag{4}$$

Finally, the total return value is calculated as:

$$RV = \frac{\sum DIFF_X}{10} \tag{5}$$

where X ranges across all evaluation criteria.

4. Adding the Genetic Code to the Population

Once a genetic code's fitness has been determined, the genetic code is inserted into the population by the population's offerMember() function. This function takes a genetic code as a parameter and inserts it into the population. If the population is already at maximum size then offerMember() removes the oldest member of the population before inserting the new genetic code.

B. Hypothesis

Quantitatively, the performance of the program in generating "better" compositions is easily measured by tracking the average fitness of the population of compositions, where this fitness is determined by user preference selections. However, this does not address the qualitative aspect as judged by a human listener. To address this, subjects were asked to rate the initial composition generated by the program on a scale from 1 to 10, and also to rate the evolved composition, also on a scale from 1 to 10. It was hypothesized that human listeners would rate the evolved composition more highly than the initial composition. The following section explains how both the quantitative and qualitative aspects of the program's composing abilities were tested.

III. EXPERIMENTAL DESIGN

To test both the quantitative and qualitative aspects of automated music composition, subjects were interviewed about their musical preferences. These subjective preferences were used to adjust program parameters so that the learning portion of the program could evolve toward styles which the user liked.

Ten evaluation criteria were used from which the program learned. These were:

- Tempo (TM)
- Tonality (TN)
- Time Signature (TS)
- Number of Instruments (NI)
- Type of Instruments (TI)
- Pitch Variation (PV)
- Note Duration Variation (NDV)
- Rhythmic Complexity (RC)
- Scales Used (SU)
- Percent of Jazz Chords (PJC)

The first of these were asked of the subjects directly. For tempo, subjects were asked if they preferred music that was slow, medium or fast, or some combination of these. (The program allows more than one tempo to be included in a composition.) Next, subjects were asked about tonality. The options for these are major, minor, or atonal (a combination of major and minor keys). If subjects were unclear on the meaning of these terms, they were phrased as generally happy, sad, or a combination.

The remaining criteria were inferred from the subjects' preferred musical genre. The mapping from musical genre to criteria is shown in Table 1. As with tempo, the program handles multiple values for time signature and type of instrument.

Table 1: Musical Genre to Criteria Mapping

Genre:	TS	NI*	TI	PV	NDV	RC	SU**	PJC
Blue-grass/ Celtic	4/4 6/8	4	Banjo Bass Fiddle Guitar Mandolin Piano	.6	.6	.4	M	.2
Blues	6/8 4/4 8/8	4	Bass Guitar Organ Piano	.5	.6	.5	P	.7
Classical	All	A	Any	.75	.8	.6	M	.8
Country/ Folk	4/4 6/8 3/4	4	Banjo Bass Fiddle Guitar Mandolin Piano	.5	.5	.4	P	.05
Experi-mental	All	A	Any	.85	.85	.85	M	.9
Hard Rock	4/4 6/8 8/8	4	Bass Guitar Organ Piano	.5	.5	.4	P	.4
Indus-trial	All	4	Bass Guitar	.4	.8	.8	P	.0
Jazz	All	A	Banjo Bass Fiddle Flute Organ Piano Sax Trumpet Xylophone	.75	.8	.85	M	.95
Metal	All	4	Bass Guitar	.7	.8	.8	M	.2
Pop	4/4 6/8 8/8	4	Bass Guitar Piano	.4	.4	.4	P	.2
Punk	2/2 4/4	4	Bass Guitar	.3	.1	.1	P	.0
Rap	4/4 8/8	4	Bass Fiddle Guitar Piano	.4	.5	.6	M	.7
Soft Rock	4/4 6/8 8/8	4	Bass Guitar Organ Piano	.5	.5	.5	P	.5
Swing/ Big Band	4/4 6/8	8	Banjo Bass Fiddle Flute Organ Piano Sax Trombone Trumpet Tuba Xylophone	.7	.7	.8	M	.8
Waltz	3/4	A	Any	.5	.4	.3	M	.5

After subjects provided their preference criteria, program parameters were adjusted to reflect these choices. The initial composition based on these parameters was saved as the "before" sample, and the program was allowed to evolve until a given fitness level, either 8.0 or 9.0 on a scale of 10.0, was reached, and that composition was saved as the "after" sample. Subjects were then asked to rate each sample according to how much they liked it on a scale from 1 to 10. The fitness level and associated learning curves measure the quantitative aspect of the program's performance, while the subject ratings capture the qualitative aspect of the system.

IV. RESULTS

Ten subjects were interviewed and the program was run for each one. In all trials, the algorithm converged to an acceptable level. On average, the program rated its own fitness level on "before" compositions at 5.06, while, on average, subjects rated the initial compositions at 5.1.

The least number of generations required to converge to the acceptable fitness level was 238 generations, while the most was 2,650, with an average convergence of 1,012 generations. On average, the program rated the fitness level of "after" compositions at 8.5, while subjects rated their liking of these compositions, on average, at 6.25.

Learning curves were typical of what one would expect from a genetic algorithm. Figure 1 shows the average fitness rating as calculated by the program across all subjects. From an average starting point of rating compositions at approximately 5.0, the algorithm shows a general trend of increasing fitness with some variation, until it levels off or converges at a level of approximately 8.5.

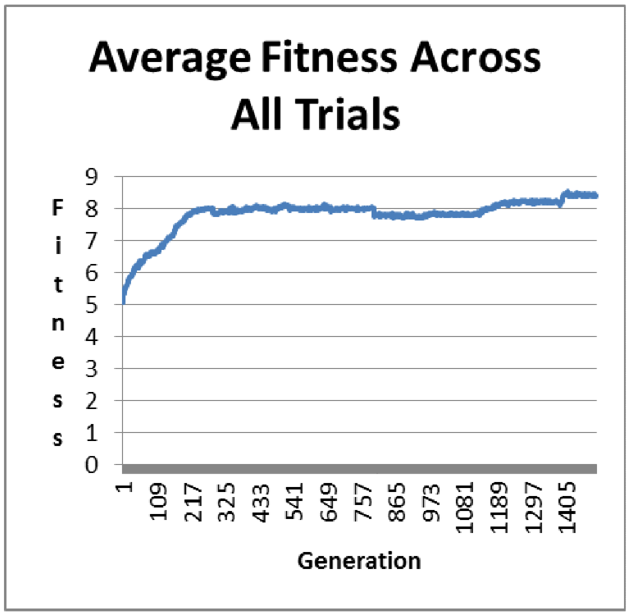

Figure 1: Average Fitness across All Trials

The correspondence between user ratings and system ratings was tested for correlation. There was a slight negative correlation between before ratings of users and the system (r= -0.68) and little correlation between after ratings given by users and by the system (r=0.49). Figure 2 shows these results graphically.

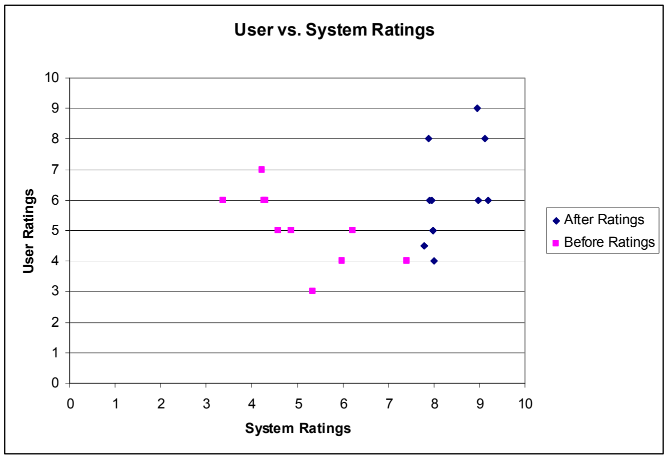

Figure 2: User vs. System Ratings

Of most interest to this research, however, is the rating given by human listeners of the compositions tailored to evolve according to their preferences. Our hypothesis stated that human listeners would rate the evolved composition more highly than the initial composition.

As shown in Figure 3, more human subjects rated the compositions higher after learning occurred than the opposite, though three subjects did rate the composition higher before learning. In Figure 3, those data points above the line of equality are those that preferred the initial composition over the learned composition, and vice versa. Note that subjects whose ratings are identical to another subject's ratings are combined into one data point.

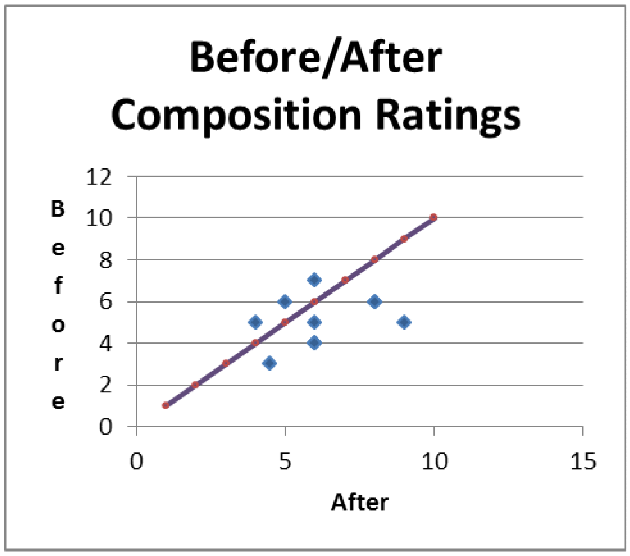

Figure 3: Before and After Ratings of Compositions

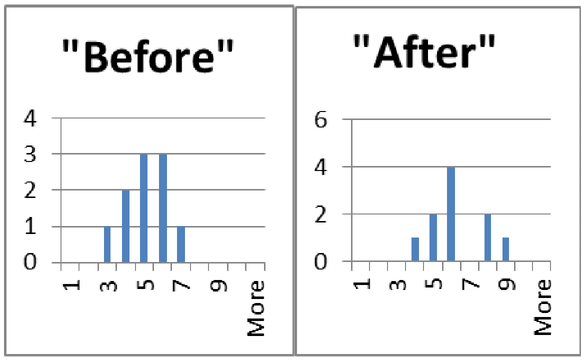

Figure 4: Frequency Distribution of the Data

The difference between "before" and "after" ratings was also tested for statistical significance. Since the hypothesis states that the expected difference is in one direction, the one-tailed test for significance was used, and it was evaluated as a paired t-test. Before and after data sets were also evaluated for approximate normal distribution (see Figure 4), and were found to meet this assumption. The difference between the two sets of ratings was statistically significant at the 95% level ($p < 0.05$).

V. CONCLUSIONS AND FUTURE DIRECTIONS

This research has demonstrated that compositions evolved from individual preference were rated by the program as closer to the given evaluation criteria via its fitness function, but more importantly, human listeners also rated the evolved compositions as better, and this difference was statistically significant. This study sets the ground work for additional investigation into the area of artificial creativity and machine learning in musical composition.

Correlation between user ratings of compositions and system ratings, both before and after learning, showed little correspondence. A possible reason for this is the method in which genre was defined. Our research used an ad hoc parameterization of different genres, relying on the knowledge of the authors. Percanella and Restaino [11] have researched a method of classifying musical genres based on fractal dimensions, providing a more objective method of definition. Future work may benefit from using the more formalized definition of genre and testing this against user perception.

During the course of subject interviews, comments made by listeners were recorded. Some of the areas of change suggested were to allow the listener to express different levels of importance for each of the criteria. That is, for one person, the choice of instrumentation may be of higher importance, while for another, the tonality (major, minor, or atonal) may be more critical. This could easily be incorporated into the program. Additional composition features that users suggested included being able to express a preference for the key in which the composition was written, the length of the composition, and changes in tempo within the composition. Again, these are changes that are easily made.

Since the results have demonstrated promise on a small subject population, future work in the area will be directed at a larger subject population, including changes to the program as discussed above.

REFERENCES

[1] Cope, David. "Experiments in Musical Intelligence." Ucsc.edu. University of California. Web. 07 June 2011. http://artsites.ucsc.edu/faculty/cope/experiments.htm .
[2] Muscutt, Keith, 2007. "Composing with Algorithms: An Interview with David Cope" Computer Music Journal 31/3 (Fall): 10-22.
[3] "How WolframTones Works." WolframTones: An Experiment in a New Kind of Music. Wolfram Research. Web. 07 June 2011. http://tones.wolfram.com/about/faqs/howitworks.html .
[4] G. Hoffman and G. Weinberg. 2010, "Gesture-based Human-Robot Jazz Improvisation" IEEE International Conference on Robotics and Automation (ICRA2010).
[5] R. Klinger and G. Rudolph. "Evolutionary Composition of Music with Learned Melody Evaluation", Proceedings of the 5th WSEAS International Conference on Computational Intelligence, Man-Machine Systems, and Cybernetics", Venice, Italy, Nov. 20-22, 2006.
[6] M. Alfonseca, M. Cebrian, and A. Ortega. "Testing Genetic Algorithm Recombination Strategies and the Normalized Compression Distance for Computer-Generated Music, Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Databases, Madrid, Spain, Feb. 15-17, 2006, 53-58.
[7] T. Maddox and J. Otten. "Using an Evolutionary Algorithm to Generate Four-Part 18th Century Harmony", Proceedings of the 1st WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE '00), Montego Bay, Jamaica, 2000.
[8] A. Santos, B. Arcay, J. Dorado, J. Romero, and J. Rodriguez. "Evolutionary Computation Systems for Musical Composition", Proceedings of the 1st WSEAS International Conference on Mathematics and Computers in Business and Economics (MCBE '00), Montego Bay, Jamaica, 2000.
[9] Goldberg, David E., Genetic Algorithms, Addison-Wesley Publishing, 1989.
[10] Adams, Ricci. "Ricci Adams' Musictheory.net." Musictheory.net. 2011. 5 June 2011 http://www.musictheory.net/.
[11] G. Percannella and R. Restaino. "Distribution-based Classification of Musical Genre Using Fractal Dimension", WSEAS Transactions on Acoustics and Music, Issue 3, Volume 1, July 2004, 123-128.

Nathan Fortier received his BS in software engineering at Montana Tech, Butte, MT, USA in 2011. He is currently a master's student at Montana State University, Bozeman, MT, USA in computer science.

He has worked as teaching assistant at Montana Tech, Butte, MT, USA for C programming and embedded systems. He has also assisted in programming for the state Transferability Initiative and has revised the computer science department website fo9r Montana Tech.

Michele Van Dyne received her BA in psychology in 1981 at the University of Montana, Missoula MT, USA, her MS in computer science in 1985, again from the University of Montana, and her PhD in electrical engineering at the University of Kansas, Lawrence KS, USA in 2003.

She was employed as a Senior Technical Programmer/Analyst for Allied-Signal Aerospace in Kansas City, MO, USA from 1985-1989, then joined Sprint Telecommunications in Overland Park, KS, USA as a Knowledge Engineer from 1989-1990. In 1990 she started an applied research company specializing in artificial intelligence, IntelliDyne, Inc., Kansas City, MO, USA, where she acted as President from 1990-2006. In 2006, she joined Montana Tech, Butte, MT, USA as an Assistant Professor in computer science. She acted as Department Head for the computer science department from 2008-2011, and is now an Associate Professor at that institution.

Dr. Van Dyne is a member of IEEE, IEEE Computer Society, ACM, and AAAI. She was the organizing member and president of ESKaMo, an expert system interest group in the Kansas City area, and steering committee member for Instructional Technology and Women in Technology interest groups of the Silicon Prairie Technology Association.