MS Project Report

Genetic Music

Ryan Becker
(rdb9723@cs.rit.edu)

Department of Computer Science
Rochester Institute of Technology
September 11, 2005

Table of Contents

Abstract

Algorithmic music composition has long been an active area of research in computer science, but the need for a human element only recently began to be more widely acknowledged. Interactive Evolutionary Computing (IEC), made popular by Karl Sims, effectively solves many high-dimension problems, like music composition, involving creative and subjective elements.

This work applies several Genetic Algorithm (GA) and Genetic Programming (GP) approaches, inspired by Karl Sims, to algorithmic music composition. The implementation of these IEC algorithms is described and their effectiveness compared.

Introduction

Algorithmic music is an incredibly active area of research which spans the fields of computer science, art, science, and philosophy. It is not a new idea, but has actually been around for as long as computers, if not longer.

History of Algorithmic Music

Algorithmic music is indeed an old concept; one could argue that even the musical dice games of Mozart [11] were really an early form of algorithmic music. A pair of dice were tossed to determine a sequence of precomposed measures, which together formed a complete piece. One even earlier technique for music composition, called "soggetto cavato" [11] and developed in the 1400s, involves mapping letters of the alphabet to notes in order to produce unique melodies (used in the Gregorian Chant). This often took the form of vowel-mapping and was used with peoples' names as well as the names of cities and towns. These simple techniques are among history's many examples demonstrating a long fascination with algorithmic music composition.

The Problem

Perhaps one of the most difficult goals of any algorithmic music system is producing something that has aesthetic appeal, which is a critical component in music – more than a nice ideal. Much of the past research in algorithmic music has focused on generating musical complexity and neglected the importance of simply sounding good. It is fairly easy to generate something that is musically complex, but generating something compelling, that tells a story, conveys an emotion, or communicates some idea, is significantly more difficult. It is this vague, subjective aspect of creativity that plagues any system that aims to generate some form of "art" or perform any task that we would generally think of as creative.

Past Approaches to Algorithmic Music

When reviewing the literature pertaining to algorithmic music, there is a wide range of approaches. Perhaps the most widely varying factor is the type and amount of human involvement. As we will see, this central issue is critical to the success of any approach.

In the early days of computer music, many algorithms were designed to be seeded with existing pieces of music [1]. These algorithms were purely statistical in nature and used Markov analysis, which simply creates tables representing the probability of a note falling at any given place in a piece. These algorithms can sometimes produce interesting variations on existing pieces of music, but the results sound much like the original piece(s) of music used to seed the Markov chains, which was often the goal anyway.

Also popular during the beginnings of research into computer music were algorithms characterized by little or no human element. This is based on the assumption that computers can generate completely original music left almost entirely to their own devices, which is by far the most ambitious goal that algorithmic music seeks. It is possible to use Markov chains in a purely generative way by skipping the analysis step and simply establishing some probabilities by filling in the transition tables, which is precisely what Hiller did in 1957 when he composed what is often considered the first original piece of music composed with a computer, the "Illiac Suite for String Quartet" [1]. This approach places more weight on the computer, and less on the human.

These types of algorithms often gave little thought to how the resulting piece of music was actually perceived, and focused more on the complexity of the resulting music. If music is to be enjoyed, this is a serious problem which must be addressed.

These are only the most notable initial approaches to algorithmic music, but many of these same ideas influenced later research, and, more importantly, thinking on human involvement did not significantly change until more recent times (1990s to the present).

Need For A Human Component

Based on the results of many years of research in computer music, it's apparent that we are incapable of coaxing computers to generate compelling music (ie. genuinely interesting, with some form or structure – not just intellectually stimulating chaos, for example) aside from human intervention, whether it be a training set, rules for composition, or some other human element [27], [19], [13].

> "No algorithm can generate meaningful music from scratch. Knowledge of the desired musical style, or a set of artificial rules, or a training set of existing music is at least needed" [19].

As soon as a human element is introduced, the chances of success dramatically improve. Still, it is difficult for a computer-generated piece of music to convey anything that the provided human elements didn't already contain, and thus, it is difficult to arrive at something "fresh" or "original." There is a clear tension here, between too much and too little of a human component.

High-Dimension Solution Space

Another of the difficulties inherent in music is the high-dimension solution space. There are simply too many variables in music for a straightforward algorithm to handle in a reasonable amount of time, if at all, which is why many of the successful algorithms limit the solution space. These variables include time, pitch, duration, loudness, meter, and tempo, among others. These variables create an infinite number of possible solutions, or pieces of music, and this high-dimension solution space is both difficult to describe and navigate.

Hypothesis

With this problem in mind, let's suppose there *were* a system that could generate interesting original music. What might it look like? The two most difficult problems are the high-dimension solution space and the need for a human component. I believe Interactive Evolutionary Computing (IEC) is the most promising technique and may have the potential to solve both of these problems, but before examining how IEC may be applied to the problem at hand, we must define Evolutionary Computing (EC) and describe its various manifestations.

Overview of Evolutionary Computing (EC)

EC describes the set of algorithms and techniques based on the theory of biological evolution. It is incredibly effective at finding solutions to problems with high-dimension problem spaces. It has been used to solve, or at least approximate solutions, to many NP problems, including the traveling salesman problem, among others. The idea has been around since the 1950s [2], but really solidified when Holland developed the Genetic Algorithm (GA) in 1962 [16]. The GA has become probably the most popular and successful of this class of algorithms. The GA's success is due, in a large part, to the publication of the classic Goldberg text [15], which made GAs palatable to the common man (or at least the common engineer), explaining in concrete terms how the GA may be applied to a variety of real-world problems.

Genetic Algorithms (GA)

In a GA, a “population” of candidate solutions is “evolved” toward better and better solutions to a problem with hopes that each successive generation will yield more “fit” solutions. The process of “evolution” involves “mutation” and “crossover” (ie. mating) operations, designed to mimic the phenomena described by the theory of biological evolution. A critical component of the GA is the fitness measure, required in order to model Darwin's “survival of the fittest.” From each generation, the most “fit” candidate solutions, or “individuals,” have a greater likelihood of persisting and reproducing, to form the next generation. One other critical component of the GA is the encoding of the solution, and the definition of the mutation and crossover operations are necessarily dependent on this encoding.

Genetic Programming (GP) [21] is a compelling variation of the GA, in which computer programs are the individuals being evolved, and fitness is determined by how well the computer program performs the desired task. This has been shown to be highly effective at solving a surprisingly wide variety of programming problems.

Interactive Evolutionary Computing (IEC)

EC seems to be the perfect solution to the high-dimension solution space problem, but what about the critical human component? IEC is a special branch of EC which includes an interactive human element, essentially providing feedback throughout the generative process. Based on our analysis of the problem, this type of algorithm would address the two most critical issues.

IEC has proven to be remarkably effective in creative or artistic processes. Karl Sims perhaps did the most to forward this method when he applied it to computer graphics [25]. He applied it to several problems, including animation, but the technique that remains most popular today, perhaps for its effectiveness and straight forward implementation, is the application to images, which provides an excellent example of IEC. For simplicity, this application will be referred to as “Genetic Images.”

The basic idea of Genetic Images is quite simple: a set of images are presented to the human and one or more favorites are chosen. These select images are then used as the basis for the next generation. Mutation and crossover operations – from GAs – are then applied to produce a new set of images. This process repeats until the algorithm produces an image the user finds appealing and decides to stop. This is essentially a GA where the human becomes the fitness function.

For something as subjective as music, there is no way to produce a standard analytical fitness function capable of effectively representing a person's individual aesthetics. Even if we could model something as complex as a person's aesthetics, it would only reflect one particular individual. What about other people, or what about when a person's aesthetics change? Aesthetics are dynamic and can change as frequently as a person's mood. It is in domains such as this that IEC works best and thus Karl Sims' successful Genetic Images provides an excellent starting point and inspiration for a Genetic Music application.

Existing Research

Applying IEC to music is not a new idea; several people have approached Genetic Music from various angles. Both standard GA [5], [9], [22], [29] and GP [20] techniques have been used and some projects have even used both together [28]. There is much to learn from what has and hasn't worked in using IEC with music, and it would be foolish not to take into consideration both the mistakes and advances of past research.

Perhaps the most serious difficulty with Genetic Music is the universally acknowledged human fatigue problem [REF]. This is a potential risk for all IEC algorithms, because the population size must be small enough for a human to evaluate in a reasonable amount of time – most Genetic Image applications use a population of around 9 individuals, for example. The problem is that when the population is reduced to numbers that small, the number of generations required to generate a satisfactory solution almost always increases dramatically. It's simply not possible for a human to evaluate millions of generations, which is not an uncommon number in many applications of traditional GAs.

Because the human fatigue problem is so critical, research in Genetic Music must either directly or indirectly focus on solving this issue. In order to lessen human fatigue, the total time required to find a good solution must be reduced. This means that either the population must converge in fewer iterations or the efficiency of the interactive human component must be improved. In other words, either the algorithm itself or the user interface (UI) must be made more efficient. It helps to keep this in mind when examining what others have done, because it is what motivates nearly all existing research.

In an attempt to solve the human fatigue problem, several people have modeled trends in user selection with a Neural Network (NN), with mostly inconclusive results [9], [20], [28]. At best, the trained NN is occasionally able to produce results almost as good as the user, but is unable to consistently do so [20]. In [28] the NN is more consistently able to produce fairly good results, greatly reducing time to converge, but the results lack diversity. Modeling human aesthetics is a very difficult problem and existing attempts seem to oversimplify, as [9] explains: "...humans listen to music in complex and subtle ways that are not captured well by simple statistical models."

Some people have focused on a particular part of music, while others have tried to do everything, including rhythm, melody, harmony, and various instruments. [5] aims to generate melodies. [29] and [12] set out to generate music consisting of multiple instruments and parts. [28] focuses on generation of rhythms. It is decidedly more difficult to handle multiple instruments and parts of music. The solution space becomes significantly larger with each added part, so often more intelligence is required for the algorithm to be successful. It is important to remember this when evaluating various studies.

The most successful algorithms to date find creative ways to narrow the solution space. For example, consider Al Biles' GenJam [5], which does Jazz improvisation well. GenJam restricts the solution space in several key ways. First, the chord progression for a piece is given, and melodic material is mapped onto tried and true jazz "scales." Since the beginning of jazz, musicians have been improvising solos based on the particular “scale” that goes with each type of chord. This mapping ensures that a “wrong note” is never played, and because it is based on techniques used by every jazz musician, it sounds right (and even hip sometimes). One other way the solution space is reduced is by limiting the number of unique notes to 14. This allows music events to be represented as 4 bit numbers, because 0 and 15 represent a rest and a hold, respectively. A rest denotes silence, and a hold causes the currently sounding note to be held through where the next note would normally sound.

The UI is still a difficult problem, and there are apparently no easy solutions. GenJam allows the user to provide positive and negative feedback on measures and phrases (a phrase consists of 4 measures) as they are played back in full, one at a time in a multi-chorus length solo, during the training phase. There are 48 unique phrases, and 64 unique measures with GenJam. Of course, because there are no wrong notes, there are likely to be many decent individuals from the beginning. This reduces the fatigue on the user because, compared to many other approaches, few individuals will tax the listener's aesthetics – a good many will likely sound pleasant.

Many approaches have little or no visual representation for music [12], [20], [22], [28]. Others [29] adopt a UI similar to that used for Genetic Images and display a visual representation, often a score. Regardless of visual representation, most of the time individuals may be played on demand. A favorite may then be selected as with Genetic Images (although it is more common to allow multiple favorites with Genetic Music). There is much room for improving the UI in Genetic Music applications.

Because the human fatigue problem is understood and acknowledged, all projects include some measures which narrow the solution space, ensuring the music complies with “good music theory,” endowing the algorithm with some idea of what is “musically interesting,” requiring a specific meter, or limiting the range of the melody, among others. While narrowing measures may be important with images, it is not nearly as critical as it is with music. Music has many more dimensions than images, and pleasing music is arguably a narrower category than pleasing images (a fairly general random approach works quite well with images). It is clear that both the type and amount of “smarts” are critical to the success of any Genetic Music algorithm.

Some Motivating Questions

There are a variety of questions that music raises, as an application domain for IEC: How is each individual represented? Can music be graphically rendered as a score or as something else? When will individuals be played and how much of each individual will be played? These questions are probably dependent on the length of each individual and the number of unique individuals which exist in each generation. Is the user an amateur musician, professional, or non-musician? How much effort can be expected from the user? What level of smarts should the GA be endowed with? On one extreme is a completely random GA, and on the other extreme is a GA with mutation and crossover operations based completely on music theory and potentially even more specific rules. In terms of scope, what are the variables or parameters in music and is it feasible to let the GA adjust them all? Finally, should a standard GA or Genetic Programming be used (ie. should we evolve music directly, or introduce a level of abstraction and evolve a music generation program, in the same way Genetic Images works)?

These are a few of the questions that motivated my project, many of which I hoped to explore and answer, in light of the patterns and trends in existing research. My goal was to implement three different approaches to Genetic Music, the first two being standard GA and GP approaches, but the final algorithm, while a GP approach, being something more original, based on mathematical curves. In fact, the concept is almost completely derived from the Genetic Images algorithm, but has never been applied quite so directly to Genetic Music. I implemented these GP and GA approaches to Genetic Music, drawing on ideas from Genetic Images and existing research in Genetic Music. I developed a working knowledge of the field of Genetic Music by implementing the principle algorithms and techniques employed, gaining a practical understanding of what works, what doesn't work, and why. It should be noted that there are no real standard approaches to Genetic Music. Thus these two algorithms, while on some level representative of existing research, include a good deal of my own ideas. There are some clear general trends, but I developed many of the details myself.

Implementation

Programming Language and APIs

All development was done in Java, for several reasons: first, good MIDI and audio support was essential. Second, good GUI support was needed to effectively implement the IEC system. Finally, it was important that the language and required libraries be readily available for the development machine, and I desired the final application to be easily deployable on many different computing platforms. This, of course, is one of the things Java does best. It seems that too often, research projects are written in obscure languages, or with obscure libraries, making it difficult to study someone's work. This project was designed to be easily accessible to anyone interested in further study, or simply curious to see for themselves how it works.

Genetic Images – Development of the IEC Framework

Before doing anything with music, a Genetic Images application was implemented, based on the original paper by Karl Sims [25]. It was important to develop the IEC framework in isolation from the complexity of the rest of the project. It was also important in evaluating whether the IEC was working or not. Genetic Images was a “safe” problem to solve with IEC, because it had been successfully implemented multiple times, and thus there were several successful projects to compare it with – there are many applet implementations across the web, for example [18], [23], [24].

Genetic Images uses a tree to represent a function, which evaluates to pixel color values. The operators are purely mathematical, consisting mostly of standard mathematical operations (addition, subtraction, multiplication, division...) and functions (sin, cos, log...). These operations, together, form a function operating on vectors representing RGB color values, which naturally makes up the leaves of the tree. The vectors at the leaves of the tree contain numerical constants or the variables X and Y. The function is then evaluated at each pixel coordinate of the image to determine a color value for that pixel.

It took a surprisingly short amount of time to develop a working system, from start to finish, thanks to the excellent Java API and particularly the Swing GUI framework. Particularly surprising was the number of interesting images generated by only the most basic of mathematical operations, addition, subtraction, multiplication, and division (see Illustration 1).

Because it turned out to be much easier to begin with a single intensity value for each pixel, support for vectors was implemented later (1.0 / X as opposed to {1 0 .5} / X). After experimenting with a number of other operators, still very simple in nature, and adding support for vectors, the complexity and depth of each image increased a great deal (see Illustration 2). Initially, only mutation was supported, by clicking on the desired image, but later, support was added for crossover, by dragging between two images (pressing the mouse button on one and releasing on another). Once users understood these two concepts, they found this method of selection quite intuitive. The development of a Genetic Images application produced some interesting results, but most importantly, it resulted in a working IEC system general enough to be applied to other problems.

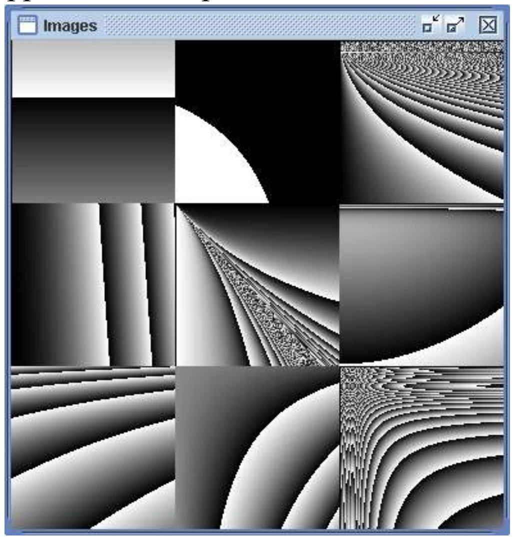


Illustration 1 - Early Genetic Images Results

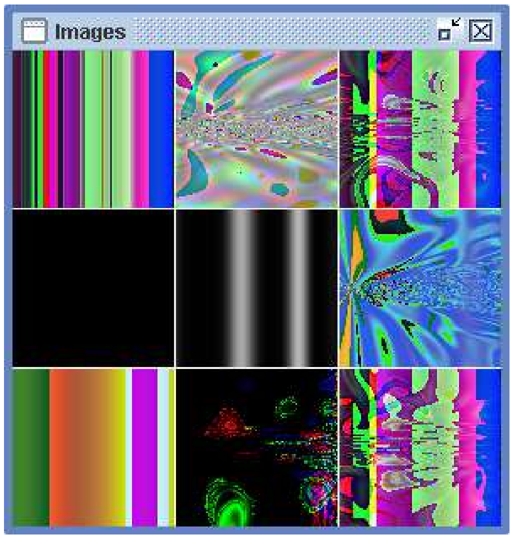


Illustration 2 - Later Genetic Images Results

Genetic Music

Common Elements

For the Genetic Music application, a software framework was built, taking full advantage of polymorphism to support any conceivable evolutionary algorithm. The basic control of the GA or GP was abstracted out, making it easy to focus solely on the algorithm. Because the goal was to implement at least three different algorithms, it was important that the framework be easily extensible.

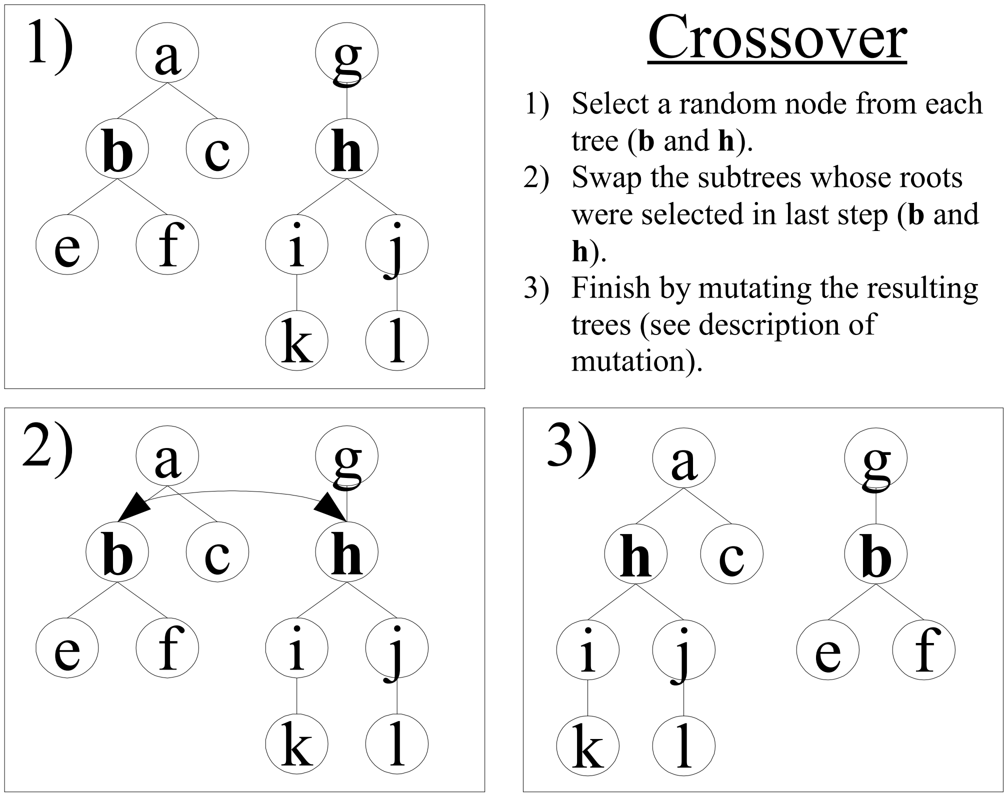
The software framework made it possible to share musical knowledge between algorithms. Specifically, a common set of musical parameters was developed, consisting of pitch, note duration, velocity, key, and mode. Each algorithm could control any number of these parameters. Because each new parameter added a whole new dimension to the problem, velocity was left out of all but one of the algorithms, and other parameters were left out entirely, the most notable being meter and tempo. A common set of musical operators were also developed, including append, repeat, reverse, shift, and blues, among others. A detailed description of each operator can be found in Appendix II.

Music was defined as a sequence of notes, containing pitch, duration, and velocity attributes.

This data structure could be described as representing a musical phrase. Using MIDI and the useful Java Sound APIs, these phrases could be rendered as audio.

GP Algorithm

The first Genetic Music algorithm implemented was a GP approach. It was not purely numerical, as with the standard Genetic Images algorithm, but included functions specifically designed for manipulation of musical phrases, like append, repeat, and reverse. The existing GP tree structure from the Genetic Images application was used with only minor changes to work with music (see Illustration 3 for an example of tree crossover and Illustration 4 for an example of tree mutation). Some operations were given a greater probability of being used than others because of the wide variety of operators – operators like "append" should be used more often than "transpose," for example. The GP tree operated on a single data type, a musical phrase. The leaves were individual notes, evaluating to single note phrases, and the branches, or interior nodes, were operators, accepting one or more phrases and returning a new phrase (see Appendix III for an example of how a GP tree is evaluated to produce a musical phrase).



Crossover

1) Select a random node from each tree (**b** and **h**).
2) Swap the subtrees whose roots were selected in last step (**b** and **h**).
3) Finish by mutating the resulting trees (see description of mutation).

Illustration 3 - GP Tree Crossover

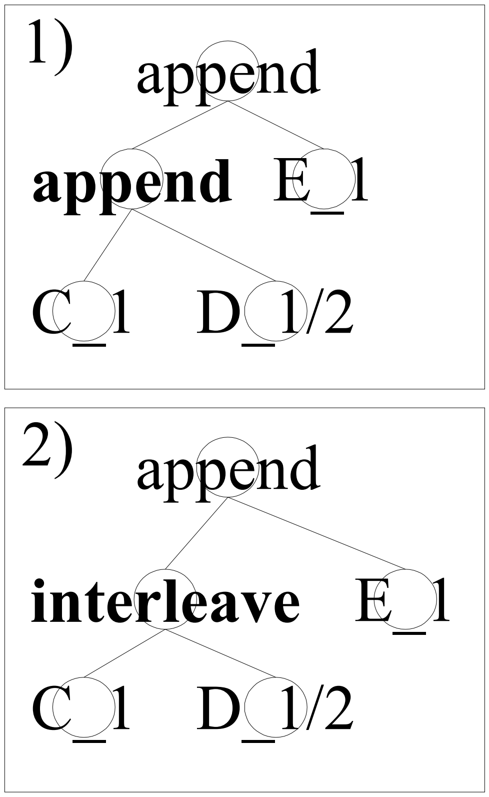


Illustration 4 - GP Tree Mutation

Mutation

1) Select a random node in the tree.
2) If it's an operator (interior node), replace it with a new random operator (**append** to **interleave**). If it's a note (leaf), replace it with a new random note.

During development of the GP algorithm, selection remained the same as in the Genetic Images application: click for mutation; drag for crossover. While the method of interaction was identical, the application visualized music and positioned the melodies in a very different manner. Each melody ran the length of the window, from left to right, and were arranged vertically – top to bottom (see Illustration 5). Each melody was visualized as a piano roll, a simple yet powerful visualization technique used by most digital music sequencers. Time ran horizontally, with green bars indicating equal measures of time (ie. the time between consecutive green bars was the same for each melody). This allowed melodies to be compressed or stretched, taking advantage of the entire available space. Notes were shown as dashes, with pitch indicated by their vertical position and duration indicated by length. For clarity, a red mark indicated the beginning of a note. Otherwise, adjacent notes of the same pitch appeared to run together as one note. Each melody could be heard by clicking the play button at its left.

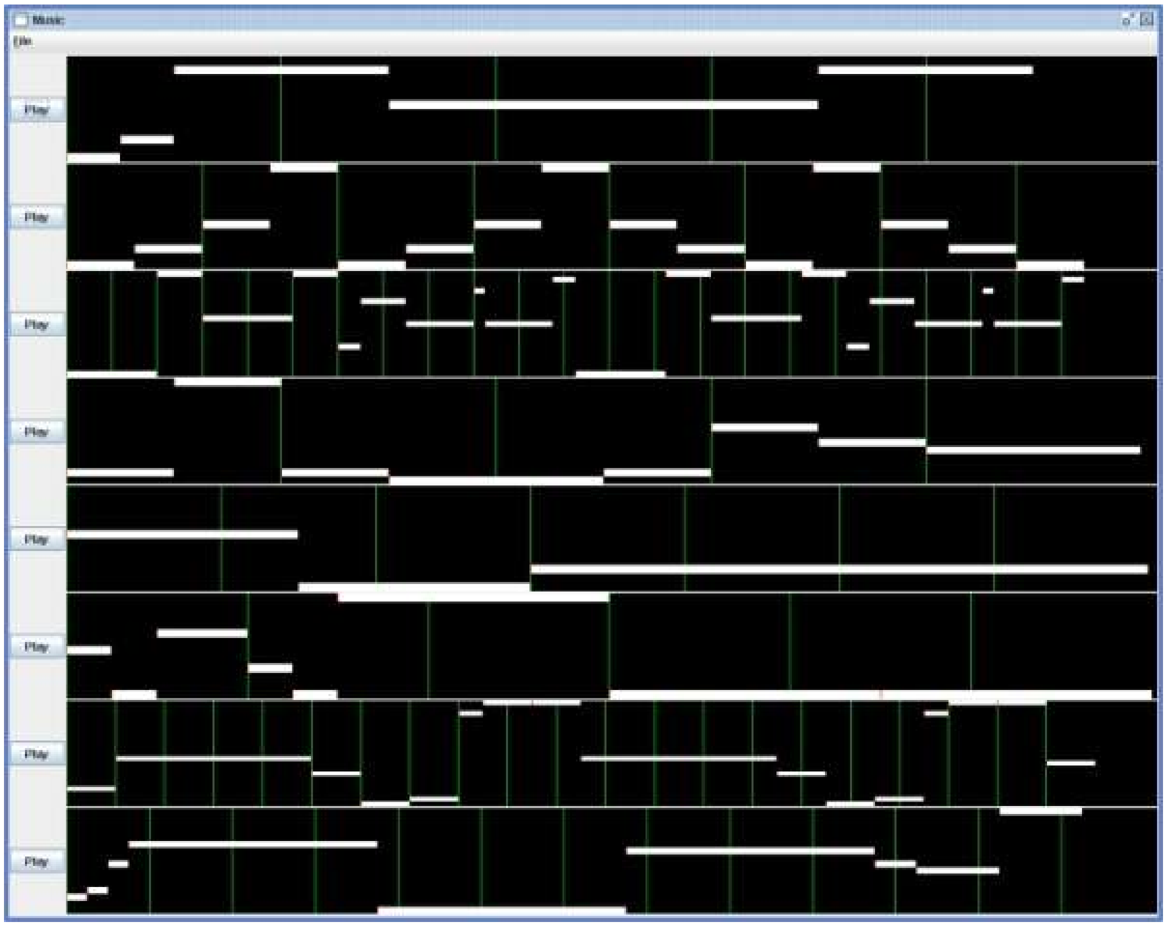


Illustration 5 - Genetic Music, UI #1

There were a couple of things wrong with this UI. First, it was a waste of space to give each melody an individual play button, particular a large one – a small icon might work better, if individual play buttons were to be considered.

Second, there was no ability to resize the window. This is not always a bad thing, but in this case, the window was very large and got in the way of other running applications.

Third, the choice of colors could be greatly improved, particularly the black background. While providing a surface on which the white notes stand out in contrast, it is considered an unpleasant color by many users. This is likely also a matter of familiarity, as common applications seldom use black backgrounds.

Finally, the method of selection, which worked well for images, was not ideal for music. There were several times where more than 2 melodies could be identified as favorites, but only 1 or 2 selections were supported by the existing selection mechanism. With images, this was not a serious problem, because the application was effective and arrived at good images in spite of this, but with music, a much larger and more complex problem space, it was essential that the interactive component be as efficient and flexible as possible.

This UI also had some positive aspects. The piano roll, for one, provided an excellent visualization of the melodies – one which could be understood by musician and novice alike. The piano roll made it possible to very quickly identify some qualities of each melody, before even listening to them. At the very least, similar melodies could be identified, potentially requiring fewer to be heard.

The other two algorithms were implemented as the UI continued to evolve (see Illustration 6). This was the beauty of a single UI. New colors were chosen and the selection mechanism fundamentally changed. Melodies could be selected (highlighted) by clicking on them (see Illustration 8 for an example, where the bottom-most melody is selected) and CTRL could be used to select more than one melody. A single "Evolve" button was added, which performed mutation or crossover, depending on the number of melodies selected. The "Play" button would play a melody if only one was selected. Pressing the "Stop" button, selecting and playing another melody, or evolving a new generation would stop playback. A single melody could be written to a MIDI file by pressing the "To MIDI" button. The current set of melodies could also be saved to a file, to be loaded at a later time. One of the most helpful new features was an "undo" command in the "edit" menu, which kept several past generations in the undo history, supporting several levels of undo. Undo not only protected against mistaken clicks, but also helped avoid elitism (Sometimes all individuals start to look the same with an evolutionary algorithm, because one "elite" individual becomes especially fit compared to others, eliminating population diversity. The "undo" command could get back to a point where other individuals still exist.).

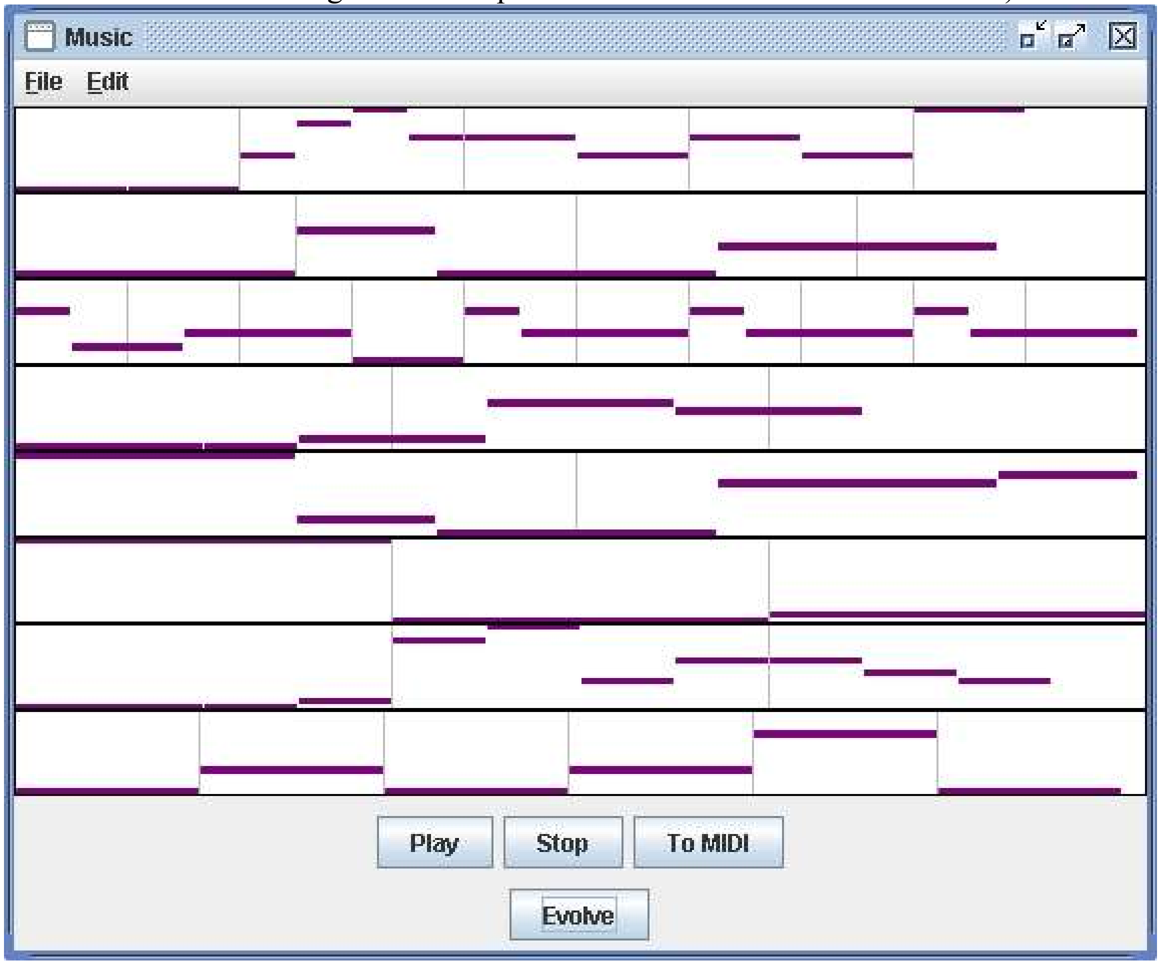


Illustration 6 - Genetic Music, UI #2

Visually, the background was changed to white and note velocity was indicated by color – red being loud and blue being soft (purple being between red and blue, representing medium loud). Rather than a red mark indicating the beginning of each note, notes were simply separated by a space, so that no two notes could ever run together. The application window could also be resized, automatically scaling all melodies, as needed. The new UI was a huge improvement over the original, and proved to be much easier to view and interact with, greatly reducing the human fatigue problem. It was in this context that the other two algorithms were implemented.

GA Algorithm

The genome in the GA algorithm was the same common phrase data type used in some form or another in all algorithms. Crossover and mutation operators also took advantage of previous work, making use of the phrase manipulation operators used as interior nodes in the GP tree. Operators were categorized as either mutation or crossover operations, depending on the number of arguments expected by each. Mutation operators were all those operating on a single phrase, while crossover operators were those operating on two. Mutation and crossover, then, consisted of simply choosing a random operator from the appropriate category and applying it.

I expected the GA approach to be very successful, but was disappointed to find that the melodies lacked something, seeming random, with no common unifying thread. What they lacked was the deeper structure and continuity that the GP algorithm provided through its abstract tree representation.

Curve Algorithm (GP)

The curve algorithm was fundamentally different, in that it was closest to the original Genetic Images algorithm. It used the exact same functions and operators from Genetic Images, but because these mathematical functions were real valued functions, function values were rounded in order to map onto discrete pitches, durations, and velocities, which took the place of red, green, and blue color values. In essence, the curve algorithm evolves three functions, whose values evaluated over time give the pitch, velocity, and duration of each note (see Illustration 7 for an example of how this works). Incidentally, this was the only algorithm where velocity, or loudness, was given an equal standing alongside pitch and duration.

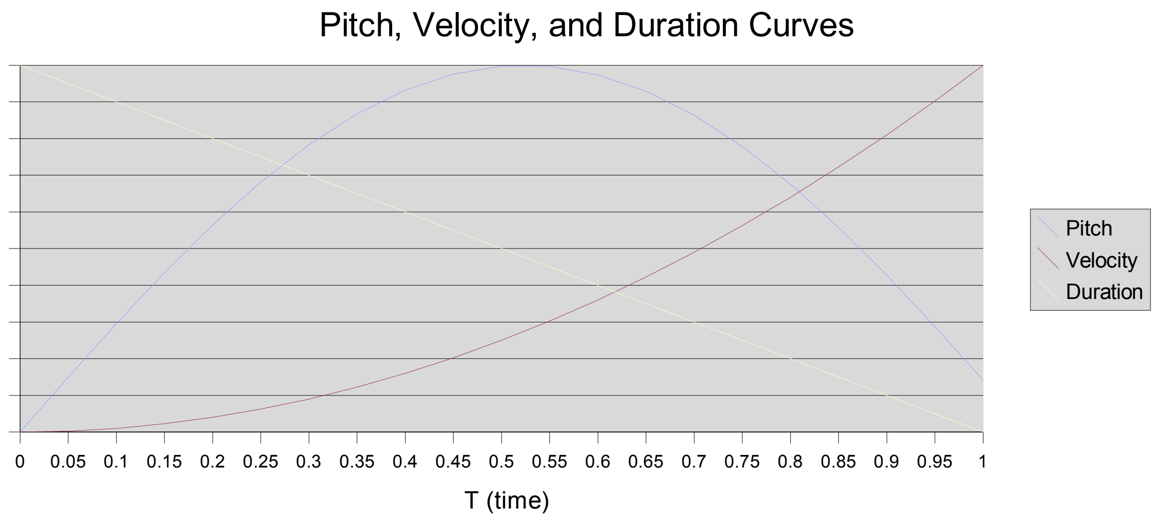


Illustration 7 - In this example, Pitch (blue) rises and falls, like part of a sine curve, velocity (red) slowly increases, at an increasing rate, and note duration (yellow falls steadily. This will result in a melody that gets louder while notes get faster and faster (shorter and shorter). The melody itself rises quickly at first, moving from low to high notes, then gently levels off before diving back down to the low notes.

The only two differences between this algorithm and Genetic Images lay in the interpretation of the function results and the specific variables used in the function trees – T (time) replaced the variables X and Y. While functions in Genetic Images were evaluated over 2 dimensional space (X and Y), functions in this application were evaluated over 1 dimensional time (T). With a solid implementation of Genetic Images to begin with, it was a simple matter to apply the same algorithm to music. It was decided that each function would be evaluated as T went from 0 to 1, where each musical time unit (between bars) was equal to .05. It was difficult to decide on this range, just as it was difficult to know how to generate discrete pitches and durations from continuous functions.

As with the GA algorithm, I expected the curve algorithm to yield better results, but again, was disappointed to find that the functions, which worked so well for images, failed miserably when applied to music. Resulting melodies, at their best, rose and fell like sine waves and parabolic functions, but completely lacked surprise and mystery – melodies were in no way compelling (see Illustration 8). There was no story, only very predictable curves. The only compelling aspect of the melodies was velocity, or volume, which naturally rises and falls according to curves in most music, unlike note pitch and duration. The other problem with this algorithm was the inherent fact that the three musical parameters, pitch, duration, and velocity, had no interrelationships – each was determined independently of the other. This algorithm highlighted the importance of these interrelationships, by demonstrating what happens when left out entirely.

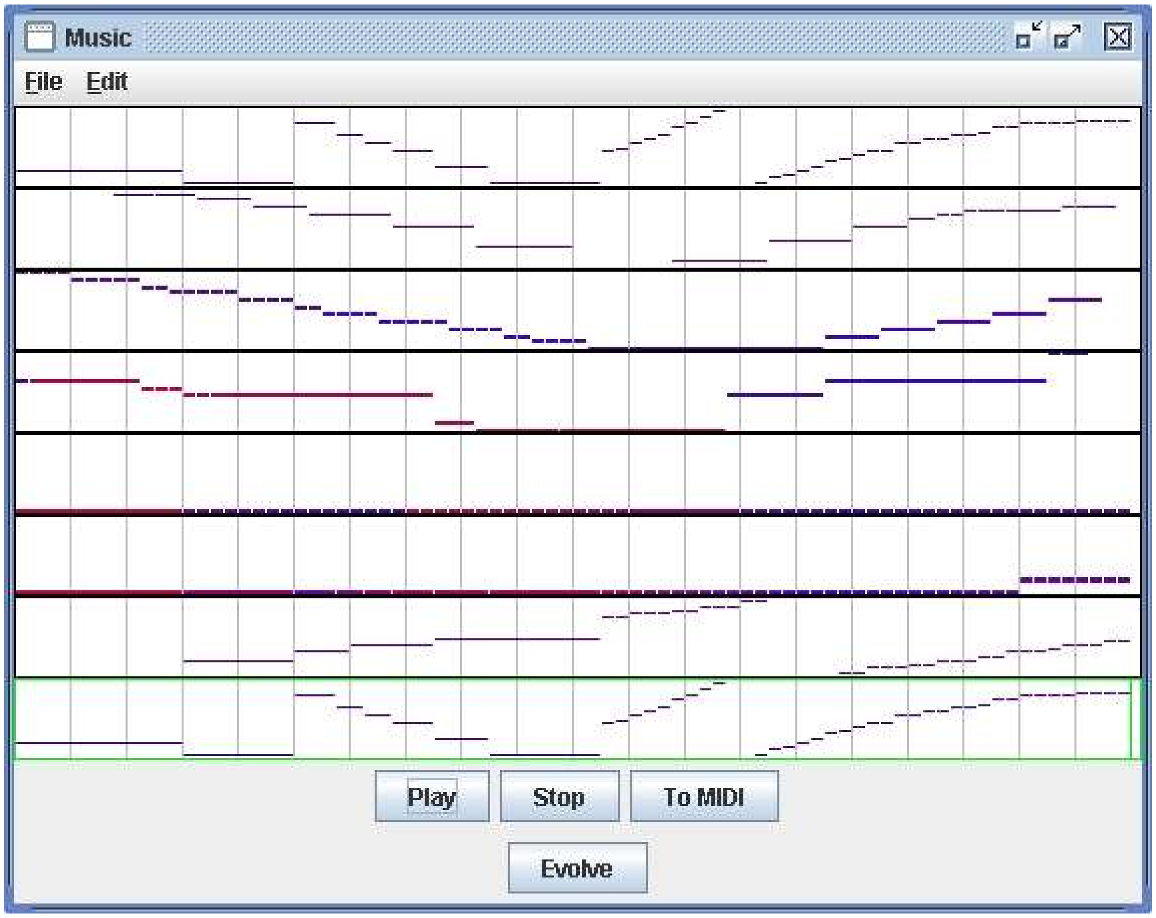


Illustration 8 - Curve Algorithm Results

While this algorithm failed in many ways, it should be emphasized that it handled velocity very well. Even though pitch and duration were seldom interesting, velocity was frequently appealing, and I found myself wishing the pitch and rhythm did it justice.

Extending the UI for the GP Algorithm

Because the GP approach was most successful, by far, I decided to extend the existing common UI, exploring the user interaction with this particular algorithm. I noticed that sometimes all but one small part of a melody was pleasing, and yet there was no way to extract the good parts and remove the bad. There needed to be a way to target feedback to particular parts of the GP tree. I considered how, given the good or bad part of a melody, the corresponding part of the tree could be identified, but realized that this was impossible. Given that many interior nodes of the tree contained only one child and considering that all operators were non-trivial it was impossible to make a simple decision at each node. There would need to be logic for each type of operator, and even then it would not work because of operators like “interleave,” which takes two phrases and alternates notes from each phrase. Suppose the bad section contained two notes. After this operation, they would be separated by a single, potentially good, note.

Eventually, I decided to expose the underlying tree structure when, in talking about Genetic Music with people from all types of backgrounds, technical and nontechnical alike, it became clear that people were capable of understanding more than might at first be expected. I implemented a tree control, which would lie just to the left of the melodies, and provide access, if desired, to each individual node (see Illustration 9). The root of the tree is a “Musical Phrases” node, which simply contains the root nodes of each musical phrase, which can be expanded to show child nodes, which in turn can be expanded to show their children, and so on, all the way down to the leaves. The tree view was designed specifically to support mutating, deleting, and copying subtrees as new phrases, but there were other operations that readily suggested themselves. These included adding a new random phrase, explicitly setting an operator or note, inserting a completely new node in the tree, removing an individual node, and replacing a subtree with a copied subtree.

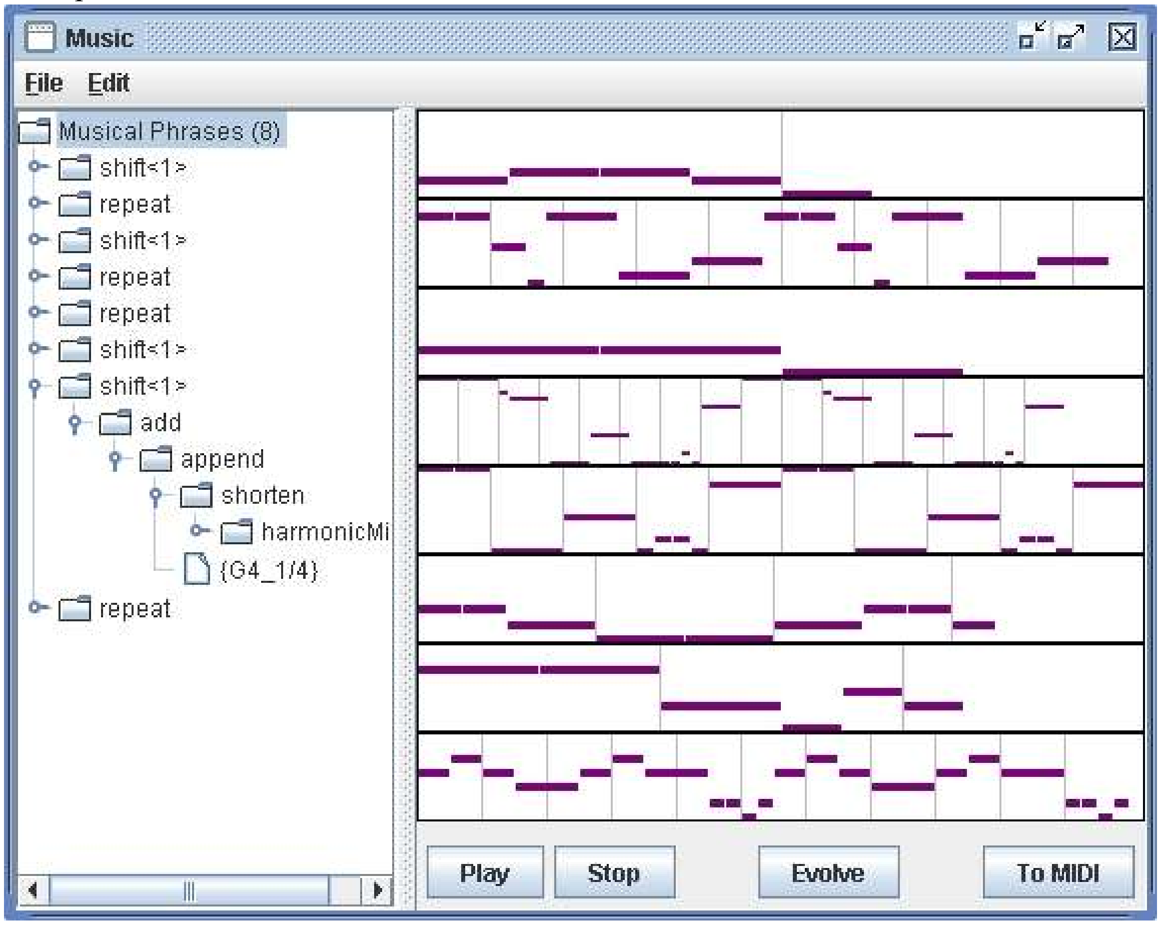


Illustration 9 - Genetic Music, UI #3

Because the tree likely provided more control than many users desired or would even know what to do with, it was still possible to interact with the application as before, selecting pleasant melodies and clicking “Evolve.” Those users who desired more control could access the tree structure and make changes as they saw fit.

With a GP algorithm, there is no way to allow more specific control of phrase evolution, without providing access in some way or another to the actual GP tree. It is hoped that further research will yield better ways to allow users to interact with the tree, hiding many of the more tedious details.

Results

While the previous section gave a preliminary indication of the results of each algorithm, the proof is in the music. All files can be found under ".\Demo\Genetic Music\Examples."

GP Algorithm

The GP algorithm was the most successful, yet more than half of the time, the first random generation provided nothing particularly good, forcing the user to either start over or perform some kind of arbitrary selection. Once a reasonably good melody or two appeared, evolution of better and more interesting melodies was not particularly difficult. This should not be a surprise, given the size of the solution space, and the fact that only 8 melodies are included in each generation. Given these results, it is understandable why other research [9], [20], [28] has explored learning algorithms and other techniques for aiding the user in selection, presenting to the user only solutions which pass certain automated tests for fitness (consider the algorithms which use NNs to model user selection). It is clear that any improvement in this area would improve the user experience in several ways; fewer unpleasant melodies would need to be endured, and pleasant melodies would appear after fewer evolutions.

File: gp1_3.midi (gp1.gmp)
Generation #: 3
Genome: (blues (append (shorten (retrograde (add (shorten (append (rotate<-1> (repeat (add (shift<-1> (rotate<1> (shorten (append {C#4_1} {E4_1/2}))))))) {G4_1/2}))))) (tiePitches (append {G4_1/2} (add (major {C#4_1}))))))

File: gp2_5a.midi (gp2.gmp)
Generation #: 5
Genome: (blues (append (shorten (repeat (append (rotate<1> (retrograde (harmonicMinor {C4_1/4}))) (rotate<-1> (lengthen (append (append (shift<-1> (shorten (shift<-1> (rotate<-1> (append (tiePitches (blues {D4_1})) (tiePitches (append {D4_1/2} {A4_1}))))))) (append (shorten (append {G4_2} (tiePitches {G#4_1}))) {C4_1/4})) {D#4_1/2})))))) {C4_1/2}))

File: gp2_5b.midi (gp2.gmp)
Generation #: 5
Genome: (blues (append (shorten (repeat (append (rotate<1> (retrograde (harmonicMinor (blues {D4_1})))) (rotate<-1> (lengthen (append (append (shift<-1> (shorten (shift<-1> (rotate<-1> (append (shift<-1> (blues {D4_1})) (tiePitches (append {C4_1/4} {A4_1}))))))) (append (shorten (append (shift<-1> (shorten (rotate<-1> (rotate<-1> (append (add (major (repeat (append (rotate<1> (retrograde (harmonicMinor (blues {D4_1})))) (rotate<-1> (lengthen (append (append {D4_1} {F4_1/2}) {D#4_1/2}))))))) (wholeTone (shorten (blues (append (append (shift<-1> (shorten (shift<-1> (rotate<-1> (append (shift<-1> (blues {D4_1}))))))) (append (append (harmonicMinor (retrograde (harmonicMinor (blues {D4_1})))) (rotate<-1> (lengthen (append (append {D4_1} {F4_1/2}) {D#4_1/2})))) {D#4_1/4})) {G#4_1/4})))))))) (append (shorten (append {G4_2} (tiePitches {G4_1}))) {C4_1/4}}))) {C4_1/4})) {D#4_1/2})))))) {C4_1/2}))

GA Algorithm

On the spectrum between chaotic and predictable, the GA algorithm produces melodies falling decidedly on the side of chaos. This would not be the case if the results were formed around some existing musical structure, as is the case with GenJam, but the GA algorithm offers very little structure itself. Resulting music sounds like a series of independent notes, rather than a connected whole. Also, it was more difficult to find continuity between each successive generation than it was with the GP algorithm, which clearly demonstrated a connection between generations. Crossover could be better defined, with operators that more accurately perform crossover, but without any imposed structure, it will be unable to achieve the same level of continuity and structure as the GP algorithm.

File: ga1_5a.midi
Generation #: 5

File: ga1_5b.midi
Generation #: 5

File: ga2_6.midi
Generation #: 6

Curve Algorithm

As was previously mentioned, the curve algorithm was disappointing, producing a predictable sequence of notes, often resembling scales, if anything. To counter this, the time range over which the functions were evaluated was increased, effectively under-sampling, but the melodies became chaotic. Yet, when the functions were adequately sampled, their sterile mathematical nature was clearly apparent. An effective time range, where these two extremes balanced, could not be found.

File: curve1_2a.midi
Generation #: 2

File: curve1_2b.midi
Generation #: 2

File: curve2_1.midi
Generation #: 1

Conclusions

Perhaps the clearest conclusion that can be drawn from this research is that a simple, sterile algorithm, devoid of human elements, cannot succeed in producing music humans will enjoy. It is simply not possible to endow an evolutionary algorithm with a minimal set of operators, inject little or no human knowledge or intelligence, and expect it to come up with a reasonably fit population in a reasonable amount of time. The number of possible solutions is too large and the number of possible fit solutions too small. This is especially apparent in this application, which only aims to evolve melodies. Consider an application where not only melodies, but harmony, instrumentation, and overall song structure are evolved.

While an algorithm completely free of human elements is not effective, it is important to realize the power of the GP approach and what makes it considerably more effective than the others. The ability to represent and maintain some type of higher level structure is critical, both to the evolutionary process, because it provides continuity, and to the actual sound of the resulting melodies. Form and structure are essential elements of good music, and GP successfully models this, at least at the melody level. It is expected that the GP approach could be extended to create structure at the song level, and everywhere in between. I believe that the GP approach is successful, is that it's higher level structure has similarities to how musicians think about music.

It was surprising to find that people were capable of comprehending much of this problem at a fairly low level. The important thing was to frame the problem in terms they understood. This realization is intriguing, and suggests that the UI could be one of the most important components of an evolutionary system, of equal importance with the algorithm itself. There is a clear tension between simplicity and control in the UI. They are inversely related – when one increases the other decreases. Even so, there is room to improve both UI simplicity and functionality without sacrificing either. The key lies in understanding the users and the tasks they want to perform, and describing the problem in a way appropriate to the user and the desired tasks.

I had hopes that little human intelligence would be required in the core algorithm, and that human selection of fit individuals would be sufficient to yield pleasing melodies in a short amount of time, in spite of a good deal of contrary evidence from the past research of other like minded people. While this goal of the project was not met, there were still many interesting results worth examining and new questions raised that have yet to be answered. Throughout the course of the project, each question answered led to the discovery of hundreds more. This is indicative both of the size and freshness of the problem. There are many unopened doors.

Future Work

Several possibilities for future work suggest themselves, particularly for the GP algorithm.

The UI could be improved a great deal. There are many ways it might be further refined and abstracted from the underlying GP implementation. The delicate balance of simplicity and control is a complex and interesting area of study, and there is much room for improvement in this application and in other similar applications. As was mentioned, an increased level of control can be given to the user if the problem can be described in a form more easily understood by the user. I am convinced that many algorithmic music projects fail because the developer or developers are simply not knowledgeable, or do not spend a significant amount of time on good UI development.

The musical structures and operators used in this application clearly could be extended. Currently, the operators operate on single line melodies with no rests. This is, at best, a limited model for music, which ought to contain elements of harmony and more complex rhythmic structures, as well as operators that structure music at various levels, all the way up to the song hierarchy level.

Harmony is important in driving a piece forward, giving it direction and flow. It also creates a richness and depth that is completely lacking in an isolated melody. Harmony has been approached from two extremely different angles, by both musicians and algorithmic music programs alike. The question is whether to build a melody on top of a harmonic progression, or build a harmonic progression around a melody. These approaches may not seem so different, at first glance, but they are really quite different in how one thinks about both melody and harmony, shaping and effecting nearly every aspect of composition. GenJam is an example of an approach that shapes melody around an existing harmonic progression. It would be interesting to go the other direction and shape a harmonic progression around an existing melody, effectively deriving a reharmonization. Another algorithm would aim to evolve both melody and harmonic progression. It could be interesting to explore whether it is more promising to evolve melody first, harmony first, or co-evolve both. It should be noted that this has all been done before, but it could be interesting to do it in the context of this application (specifically, the UI).

Regarding rhythm, it is important to have patterns on many levels, through an entire piece as well as through a single phrase. It is also critical to recognize that silence is just as important and powerful in generating rhythm as sound. The existing system has no high level representation for rhythm, or any representation, for that matter, and resulting music is thus lacking any strong rhythmical pattern or structure. It could stand to benefit greatly from even simple concepts of repetition and quantization.

Perhaps both harmony and rhythm could be implemented with a more general musical representation and operators that handle these concepts. There are rules for developing harmony, which many composers follow. It might be possible to add operators that encapsulate this knowledge. Of course, from a computer science theory perspective, it would be more interesting if the algorithm were able to evolve this knowledge itself, but providing smarter operators is a definite possibility.

If this approach to Genetic Music is to ever be more than a toy system that creates short melodies, operators must have a granularity. There must be operators that manipulate individual notes *and* operators that manipulate overall song structure. There are many granularity levels that might be considered. It is likely that this would somewhat improve the algorithm's efficiency, because existing operators are sometimes wasted, depending on where they appear in the GP tree, but it is also possible that this would result in a simplification limiting the variety of the resulting music. I expect this is another area where delicate balancing is required.

While this work has failed to seriously address velocity, it is a core element of music. It might be possible to create a hybrid approach between the GP and curve algorithms, where velocity is determined with the mathematical operators based on those used in Genetic Images, and the remaining musical elements are determined with the musical operators of the initial GP approach. This brings up an interesting concept. Suppose the GP “language” were made more complex, given more data types – in this case, numbers for velocity, and phrases for pitch and duration. This might be an interesting path of research.

These are merely a handful of ways in which this research could be extended or built upon. There are many angles that have yet to be explored, and with something as subjective and subtle as music, each person will bring something new and unique to the field. There are as many philosophies about music as there are people, and the beauty of music is that each of these philosophies is valid, because music is in the ear of the listener.

Bibliography

1: Assayag, Gérard. Computer Assisted Composition Today. 1st SYMPOSIUM ON MUSIC AND COMPUTERS. "Applications on Contemporary Music Creation, Aesthetic and Technical aspects". CORFU. October 1998.
2: Back, Thomas, Hammel, Ulrich, and Schwefel, Hans-Paul. Evolutionary Computation: Comments on the History and Current State. IEEE Transactions On Evolutionary Computation, Vol. 1, No. 1. 3-17. April 1997.
3: Becker, Ryan. Algorithmic Music: An overview of past research and a look to the future. Research Topic for Virtual Theatre (Spring 2004), Department of Computer Science, Rochester Institute of Technology. May 2004.
4: Becker, Ryan. Interactive Evolutionary Computation: With a focus on Genetic Music. Research Topic for Computer Animation: Algorithms & Techniques (Winter 2004), Department of Computer Science, Rochester Institute of Technology. February 2005.
5: Biles, John A.. GenJam: A Genetic Algorithm for Generating Jazz Solos. Proceedings of the 1994 International Computer Music Conference, ICMA, San Francisco. 1994.
6: Biles, John A.. Autonomous GenJam: Eliminating the Fitness Bottleneck by Eliminating Fitness. Proceedings of the GECCO-2001 Workshop on Non-routine Design with Evolutionary Systems.. 2001.
7: Biles, John A.. GenJam in Transition: from Genetic Jammer to Generative Jammer. Generative Art. 2002.
8: Biles, John A.. GenJam: Evolutionary Computation Gets a Gig. Proceedings of the 2002 Conference for Information Technology Curriculum, Rochester, New York, Society for Information Technology Education. September 2002.
9: Biles, John A., Anderson, Peter G., and Loggi, Laura W.. Neural Network Fitness Functions for a Musical IGA. Proceedings of the International ICSC Symposium on Intelligent Industrial Automation and Soft Computing. 1996.
10: Burns, Kristine H.. Algorithmic Composition. http://eamusic.dartmouth.edu/~wowem/hardware/algorithmdefinition.html. February 2004.
11: Burns, Kristine H.. History of electronic and computer music including automatic instruments and compositional machines. http://music.dartmouth.edu/~wowem/electronmedia/music/eamhistory.html. February 2004.
12: Felice, Fabio De, Abbattista, Fabio, and Scagliola, Francesco. GenOrchestra: An Interactive Evolutionary Agent for Musical Composition. Generative Art. 2002.
13: Garnett, Guy E.. The Aesthetics of Interactive Computer Music. Computer Music Journal, Vol. 25, No. 1. 21-33. March 2001.
14: Gena, Peter. Lejaren Hiller (1924-1994). http://www.artic.edu/~pgena/lhobit.html. February 1994.
15: Goldberg, David E.. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA. 1989.
16: Holland, John H.. Outline for a Logical Theory of Adaptive Systems. Journal of the ACM (JACM), Vol. 9, Issue 3. 297-314. July 1962.
17: Hudak, Paul and Berger, Jonathan. A Model of Performance, Interaction, and Improvisation. Proceedings of International Computer Music Conference. Computer Music Association. 1995.
18: Huxtable, Jerry. Genetic Art. http://www.jhlabs.com/java/art.html. January 2001.
19: Järveläinen, Hanna. Algorithmic Musical Composition. Seminar on content creation,

Telecommunications software and multimedia laboratory, Helsinki University of Technology. April 2000.
20: Johanson, Brad and Poli, Riccardo. GP-Music: An Interactive Genetic Programming System for Music Generation with Automated Fitness Raters. Genetic Programming 1998: Proceedings of the Third Annual Conference. 181-186. 1998.
21: Koza, John R.. Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Technical Report STAN-CS-90-1314, Stanford University Computer Science Department. June 1990.
22: Moroni, Artemis, Manzolli, Jônatas, Zuben, Fernando Von, and Gudwin, Ricardo. Vox Populi: An Interactive Evolutionary System for Algorithmic Music Composition. Leonardo Music Journal, Vol. 10. 49-54. 2000.
23: Mount, John. geneticArt IV. http://mzlabs.com/gart/g4.html. May 2003.
24: Saunders, Robert. Evol. http://www.arch.usyd.edu.au/~rob/java/applets/genart/GenArt.html. August 2001.
25: Sims, Karl. Artificial Evolution for Computer Graphics. Computer Graphics, Vol. 25, No. 4. 319-328. July 1991.
26: Takagi, Hideyuki. Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation. Proceedings of the IEEE, Vol. 89, No. 9. 1275-1296. September 2001.
27: Todd, Peter M. and Werner, Gregory M.. Frankensteinian methods for evolutionary music composition. Musical Networks: Parallel distributed perception and performance, N. Griffith and P. Todd, eds. Cambridge, MA: MIT Press. 1998.
28: Tokui, Nao and Iba, Hitoshi. Music Composition by Means of Interactive GA and GP. Proceedings of the 2001 IEEE Systems, Man and Cybernetics Conference. 2001.
29: Unemi, Tatsuo and Senda, Manabu. A New Music Tool for Composition and Play Based on Simulated Breeding. Proceedings of Second Iteration. 100-109. 2001.

Appendix I – User Documentation

Genetic Images

Installation / Requirements

Requires version 5.0 or later of the Java Runtime Environment (JRE 5.0).

Installation is simple. Everything is in one self-contained jar file, which may be executed with "java -jar Images" from the command line, or simply double-clicked in Windows.

Use

As soon as the program is launched, 9 randomly generated images will be displayed in a 3x3 arrangement. A favorite may be chosen by clicking on an image. Two may be selected by dragging from one to the other (pressing and holding the mouse button on one, dragging the cursor to another, and releasing the mouse button). As soon as a 1 or 2 favorites have been selected, 9 new images are generated, based on the selection. More images may be generated in this manner as long as desired.

Genetic Music

Installation / Requirements

Requires version 5.0 or later of the Java Runtime Environment (JRE 5.0). The Java Media Framework (JMF) with the best quality MIDI samples is highly recommended. This is usually an option during the installation procedure.

Installation is simple. Everything is in one self-contained jar file, which may be executed with "java -jar Music" from the command line, or simply double-clicked in Windows.

Use

When the program begins, a dialog appears, allowing the user to select the desired algorithm (GP, GA, or Curve). After a choice is made, 8 random phrases are generated according to the chosen algorithm. Phrases may be played, stopped (if playing), evolved, or saved to a MIDI file with the buttons at the bottom of the window. These actions are applied to all selected phrases, with the exception of play, which only works with a single selection. Phrases are selected by clicking on them. Selected phrases are outlined in green. Multiple selections are made by CTRL-clicking, which appends a phrase to existing selections.

<u>**File Menu**</u>

New – Creates a new random set of phrases, replacing any existing phrases.

Open – Opens a saved generation (set of phrases).

Save – Saves the existing generation to a file. The Genetic Music application recognizes .gmp files (Genetic Music Population). If the phrases were never saved in either this generation or past generations, the application will prompt for a filename; otherwise, the application prompts for a filename.

Save As – Saves the existing generation to a file, always prompting for a new filename.

Export...->to MIDI / to WAV – Exports each existing phrase to an audio file, prompting for a common filename prefix and appending sequential numbers to differentiate each melody. WAV output is not implemented.

Quit – Exits the application.

Edit Menu

Undo – Reverts to the previous set of phrases (if one exists). There are 10 levels of undo, which seems to be more than adequate in most cases.

Known Issues

There is a phrase rendering bug which occasionally causes the second half of a phrase to appear blank. It is expected that this is a boundary issue, perhaps regarding upper and lower bounds on pitch, but this is an elusive bug and this theory has never been verified. Interestingly, resizing the window (and thus the phrases) causes the rest of the offending phrase to be rendered.

Genetic Music – GP Tree Extension

Installation / Requirements

Requires version 5.0 or later of the Java Runtime Environment (JRE 5.0). The Java Media Framework (JMF) with the best quality MIDI samples is highly recommended. This is usually an option during the installation procedure.

Installation is simple. Everything is in one self-contained jar file, which may be executed with "java -jar MusicGPTree" from the command line, or simply double-clicked in Windows.

Use

Only the tree interface will be described here. See the standard Genetic Music user documentation for a description of the rest of the application.

Nearly all tree operations are performed via the context menu, usually based on the selected tree nodes. Tree nodes are selected by clicking on them. Multiple tree nodes are selected by CTRL-clicking, to append nodes to the current selection.

Context Menu

New Melody – Creates a new random melody, adding it at the botton of the list of existing melodies.

Copy As New Melody – Copies the selected subtrees as new melodies.

Copy – Copies the selected melody or subtree.

Make Root – Make the selected subtree the root of the tree, replacing the melody with this subtree.

Delete – Deletes the selected melodies or subtrees (replacing with a single note). This may also be triggered with the "Delete" key on the keyboard.

Mutate – Mutate the selected melodies or subtrees.

Change To... – Opens a submenu, allowing the currently selected operators to be changed to any other operator.

Paste – Paste the previously copied melody or subtree over the selected melodies or subtrees.

Appendix II – Detailed Description of Musical Operators

Note that every operator operates on a phrase, returning the result as a phrase and accepting any arguments as phrases. It is helpful to understand how both notes and phrases are represented, as these formats will be used in the examples below. This is what was used by the toString methods in the Java program. Notes are given as <pitch><octave>_<duration>, where duration is in terms of quarter notes. For example, "C4_1" represents a quarter note at middle C. A phrase including the notes C, D and E, the first 3 notes in the C scale, starting at middle C, where the C is a dotted eighth note, the D is a sixteenth note, and the E is a quarter note would be represented as "{C4_3/4 D4_1/4 E4_1}". When it is not important, the duration of each note is left out ("{C4 D4 E4}" for example). The number immediately after the note name is the octave. These octave numbers are standard in music theory and are, understandably, used extensively in computer music applications as well. Middle C on a piano is always C4.

append(phrase1, phrase2)

Appends phrase1 to phrase2. For example:

append({C4 D4 E4}, {A3 B3 C4}) => {C4 D4 E4 A3 B3 C4}

interleave(phrase1, phrase2)

Interleaves phrase1 with phrase2. For example:

interleave({C4 D4 E4}, {A3 B3 C4}) => {C4 A3 D4 B3 E4 C4}

addDegree(phrase1, phrase2) – unused

Essentially shifts each note in phrase1 *up* by the degree of each note in phrase2 (degree is determined according to the key of each phrase). For example:

addDegree({C4 D4 E4}, {A3 B3 C4}) => {A4 C5 E4}

Suppose the key of both phrases is C Major. Thus the degrees for phrase2 are 6, 7, and 8 or 1 (since there are only 7 degrees before the C Major scale repeats, and C is the first degree of the C Major scale). The actual degrees used internally are 0 based though, so the degrees are actually 5, 6, and 0. So we move C4 up 5 degrees (->D4->E4->F4->G4->A4), D4 up 6 degrees (->E4->F4->G4->A4->B4->C5), and E4 up 0 degrees (E4).

subtractDegree(phrase1, phrase2) – unused

Essentially shifts each note in phrase1 *down* by the degree of each note in phrase2. For example:

subtractDegree({C4 D4 E4}, {A3 B3 C4}) => {E3 E3 E4}

C4: B3->A3->G3->F3->E3
D4: C4->B3->A3->G3->F3->E3
E4: E4

shorten(phrase1)

Shortens each note. This operation originally simply subtracted the smallest allowed note duration (a 16th note), but this didn't provide enough variation, resulting in especially ackward and syncopated rhythms. In the end, each note duration was handled on a case by case basis. Anything greater than a whole note is decreased by a half note, anything greater than a half note is decreased by a quarter note, anything greater than a eighth note is decreased by an eighth note, and anything else is decreased by a sixteenth note. No note may go below a sixteenth note.

shorten({C4_1 D4_3/4 E4_1/4}) => {C4_1/2 D4_1/4 E4_1/4}

lengthen(phrase1)

Lengthens each note. This operation was also handled on a case by case basis. Anything less than an eighth note is increased by a sixteenth note, anything less than a half note is increased by an eighth note, anything less than a whole note is increased by a quarter note, and anything else is increased by a half note.

lengthen({C4_1 D4_3/4 E4_1/4}) => {C4_3/2 D4_5/4 E4_1/2}

palindrome(phrase1)

Appends a reversed copy of the phrase to itself, creating a palindrome (reads the same forward and backward).

palindrome({C4 D4 E4}) => {C4 D4 E4 E4 D4 C4}

repeat(phrase1)

Repeats the phrase, appending a copy.

palindrome({C4 D4 E4}) => {C4 D4 E4 C4 D4 E4}

retrograde(phrase1)

Or reverse... Reverses a phrase.

retrograde({C4 D4 E4}) => {E4 D4 C4}

rotate<1>(phrase1)

Rotates a phrase *forward* 1 place, shifting everything *right* and taking the last note and placing it at the front of the phrase. This operator takes an argument when it is created, which can be a positive or negative integer, and indicates how far the phrase is rotated and in which direction (-4 would rotate backward 4 places, bumping the first 4 notes to the back of the phrase).

rotate<1>({C4 D4 E4}) => {E4 C4 D4}

rotate<-1>(phrase1)

Rotates a phrase *backward* 1 place, shifting everything to the *left* and bumping the first note to the end of the phrase. This is the only other rotate operator, but, as is hinted above, there could be other rotate operators. It was decided that there should be only these 2 though, because by combining in different ways, these two rotations could immitate the behaviour of every possible rotation operator.

rotate<-1>({C4 D4 E4}) => {D4 E4 C4}

add<1>(phrase1)

This is another operator with a parameter. This one adds a note to the phrase, at the given degree offset. This specific operator will add a note 1 degree *above* the last note in the phrase.

add<1>({C4 D4 E4}) => {C4 D4 E4 F4}

add<-1>(phrase1)

This specific operator will add a note 1 degree *below* the last note in the phrase.

add<-1>({C4 D4 E4}) => {C4 D4 E4 D4}

shift<1>(phrase1) – unused

Yet another parameterized operator, shifting a phrase by degrees (which differentiates it from transposition, because it maintains the key of the original phrase). This version will shift all notes *up* by 1 degree.

shift<1>({C4 D4 E4}) => {D4 E4 F4}

shift<-1>(phrase1) – unused

Shifts all notes *down* by 1 degree.

Shift<-1>({C4 D4 E4}) => {B3 C4 D4}

tiePitches(phrase1)

Creates a "tie" of sorts between notes (a musical tie is when two identical notes are adjacent, but only the first note sounds, being held through the second note, as though only one note were written). Every pair of adjacent identical notes are merged, creating single, longer notes.

tiePitches({C4_3/4 C4_1/4 D4_1 E4_1 E4_1}) => {C4_1 D4_1 E4_2}

major(phrase1)

Places the phrase on a major scale, if not already. In this example, the phrase is in the key of C minor (the D#, while the correct pitch, should really be given as Eb, but this is how the program would represent it, because it doesn't know when something should be a sharp or flat – "#" or "b"), but the result is in the key of C Major, because the Eb has been turned into an E.

major({C4 D4 D#4}) => {C4 D4 E4}

harmonicMinor(phrase1)

Places the phrase on a harmonic minor scale, if not already. In this example, the phrase begins in the key of A Major.

harmonicMinor({A3 C#4 E4 F#4}) => {A3 C4 E4 F4}

melodicMinor(phrase1) – unused

Places the phrase on a melodic minor scale, if not already. In this example, the phrase begins in the key of A Major.

> melodicMinor({A3 C#4 E4 F#4}) => {A3 C4 E4 F#4}

blues(phrase1)

Places the phrase on a blues scale, if not already. In this example, the phrase begins in the key of A Major.

> blues({A3 B3 C#4 D4 E4}) => {A3 C4 D4 D4 E4}

wholeTone(phrase1)

Places the phrase on a whole tone scale, if not already. In this example, the phrase begins in the key of A Major.

> wholeTone({A3 B3 C#4 D4 E4}) => {A3 B3 C#4 D#4 F4}

pentatonic(phrase1)

Places the phrase on a pentatonic scale, if not already. In this example, the phrase begins in the key of A Major.

> pentatonic({A3 B3 C#4 D4 E4 F# G#}) => {A3 B3 C#4 C#4 E4 F#4 A4}

Appendix III – Evaluating a GP Music Tree

Following is a GP music tree that evaluates to a phrase from the popular children's song, “Mary Had A Little Lamb.” The words for this part of the song are “Mary had a little lamb, its fleece was white as snow.” This is a contrived example, used simply to demonstrate the GP music tree. Notes are given as <pitch><octave>_<duration>, where duration is in terms of quarter notes. For example, “C4_1” is a quarter note at middle C.

> *(append (append (reverse (append {C4_1} **(add<1> {D4_1})**)) (palindrome (append {D4_1} **(repeat {E4_1})**))) (interleave **(repeat {D4_1}) (append {E4_1} {C4_1})**))*

Add a higher degree note to {D4_1}, repeat {E4_1}, repeat {D4_1}, and append {E4_1} to {C4_1}.

> *(append (append (reverse **(append {C4_1} {D4_1 E4_1})**) (palindrome **(append {D4_1} {E4_1 E4_1})**)) **(interleave {D4_1 D4_1} {E4_1 C4_1})**)*

Append {C4_1} to {D4_1 E4_1}, append {D4_1} to {E4_1 E4_1}, and interleave {D4_1 D4_1} and {E4_1 C4_1}.

> *(append (append **(reverse {C4_1 D4_1 E4_1}) (palindrome {D4_1 E4_1 E4_1})**) {D4_1 E4_1 D4_1 C4_1})*

Reverse {C4_1 D4_1 E4_1} and append the reverse of {D4_1 E4_1 E4_1} to itself (creating a palindrome).

> *(append **(append {E4_1 D4_1 C4_1} {D4_1 E4_1 E4_1 E4_1 E4_1 D4_1})** {D4_1 E4_1 D4_1 C4_1})*

Append {E4_1 D4_1 C4_1} to {D4_1 E4_1 E4_1 E4_1 E4_1 D4_1}.

> *(append {E4_1 D4_1 C4_1 D4_1 E4_1 E4_1 E4_1 E4_1 D4_1} {D4_1 E4_1 D4_1 C4_1})*

Finally, append {E4_1 D4_1 C4_1 D4_1 E4_1 E4_1 E4_1 E4_1 D4_1} to {D4_1 E4_1 D4_1 C4_1}...

> *{E4_1 D4_1 C4_1 D4_1 E4_1 E4_1 E4_1 E4_1 D4_1 D4_1 E4_1 D4_1 C4_1}*

Placing the words of the song over the corresponding note, we have...

> ***Ma - ry had a lit - tle lamb, its fleece was white as snow.***
>
> *{E4_1 D4_1 C4_1 D4_1 E4_1 E4_1 E4_1 E4_1 D4_1 D4_1 E4_1 D4_1 C4_1}*