Artificial Life Music: An A-Life Approach to Generating Melodies Featuring Genetic Algorithms

Jacob Peck
Computer Science Department
State University of New York, College at Oswego

April 29, 2011

Abstract

This report describes an algorithmic composition program that utilizes L-systems, cellular automata, and genetic algorithms to attempt to compose melodies. A brief history of past projects in the field is given, along with a complete description of the approach taken by this program. An analysis of various abstractions in the program, a thorough discription of the genetic algorithm implemented, and discussion of results is also given.

Keywords

Algorithmic composition; cellular automata; computer music; generative music; genetic algorithims; Java; JFugue; Lindenmayer systems; Lisp; L-Systems; melody; MIDI; rule 86.

Introduction

This report details an algorithmic composition program which utilizes cellular automata, Lindenmayer systems, and genetic algorithms to generate melodies. This project was undertaken to increase understanding of all of these areas of computation and algorithmic composition, and also to start building a collection of useful, generic modules written in Lisp.

Such a project does not exist in a vacuum, so discussion of previous research and others' accomplishments is in order. Also discussed is the approach taken by this particular program to the generation of melodic lines, a brief discussion of the genetic algorithm in use, and other specifics to this program. Lastly some results are analyzed, avenues for future research are laid out, and a summary of the entire project is given.

Background

Algorithmic composition—the use of formalizable methods for musical composition—has been practiced for centuries, dating back to Guido d'Arezzo's formalized approach circa 1000 AD at turning written text into melodic lines [1]. Throughout the years, algorithmic composition has tended to become more directly mathematical and scientific, due at least in part to the rise of the computer. With Hiller and Issacson's *Illiac Suite*, algorithmic composition was brought into a new level of acceptance [1].

In recent algorithmic compostion efforts, focus has been directed towards mathematical and scientific formalisms. A number of L-System and cellular automata based approaches have been advanced, along with a few based in genetic algorithms [1, 2, 3, 4, 5, 6]. The project discribed in this report borrows from all of these approaches in a way that is unique and provides avenues for future research.

Related Research

Many algorithmic composers have done work with cellular automata. Perhaps the most notable is Eduardo R. Miranda's program `CAMUS`, short for **C**ellular **A**utomata **MUS**ic. This program utilizes coupled cellular automata running separate rulesets to generate both pitch and rhythmic aspects of chord progressions [1].

Some research has been done in the field of L-system music as well. Martin Supper offers a small example L-system music [3]. He references Hanspeter Kyburz's L-system approach to generating a piece for saxophone and ensemble, *Cells* [3]. One of the author's previous programs, `LCompose`, utilized a user-defined L-system to generate melodies [4]. One major disadvantage (although some would say it is an advantage) of this approach is that the composer has complete control over what is going to happen. One reason that this project was initiated was to overcome the limitations of `LCompose`.

A large body of works in the field of genetic algorithm music allows for pretty decent style replication. Researchers such as Bruce L. Jacob demonstrate this [5]. John A. Biles has used genetic algorithms for generating jazz solos [6].

Approach

This program uses a cellular automata selection grid layered overtop of a population of L-systems, on a one-to-one basis. These L-systems are derived from randomly generating a number of different instances to form the initial population from an alphabet entered by the user. From this population, a Darwinian approach is taken (see *Description of the Genetic Algorithm*, below) to evolve these L-systems into a more fit population from which to assign to cells in the cellular automata selection grid.

The cellular automata is a torroidal 1 dimensional grid following the "rule 86" rules for each generation.

Rule 86 is a rule system for one-dimensional cellular automata which allows for semi-chaotic behavior. It is defined by the following table:

Current state	XXX	XX-	X-X	X--	-XX	-X-	--X	---
Next state	-	X	-	X	-	X	X	-

where - denotes a "dead" cell, and X denotes an "alive" cell. The next state of the center cell is determined by its two neighbors [7]. So, for example, if the state of the world was:

```
---X-XXX----X-X-X---XXX---X-X-X---X-XXX--X-XXXXX--
```

the next generation would be:

```
--XX---XX--XX-X-XX-X--XX-XX-X-XX-XX---XXXX-----XX-
```

and so on.

Each cell also has an individual history, which is used to select which cell's L-system is applied each iteration. Cells which have been alive more often are preferred over cells that have not.

Each iteration, the entire cellular automata iterates. After this, the live cell with the largest number of "alives" in its history is chosen to retrieve the L-system from. The entire current string of data is

passed through this L-system to produce an output data string, and then it is displayed to the user. Should the user decide to continue with another iteration of the system, the entire process starts over again. This continues until the user decides to terminate the compositional phase. At this point, the composition is sent to `JFugue` to write a MIDI file as output, and the program ends.

Program Abstractions

This program contains several abstractions. The nature of Lisp allows for several conveniences in modelling the domain of this project.

Cellular automata

The cellular automata consists of two classes: `cell`, which models an individual cell, and `ca`, which models the entire world.

The `cell` class contains pointers to its left and right neighbors, a history of all previous states maintained as an ordered list, and a symbolic representation of its current state. This class is extended by the class `lcell`, which simply allows a cell to also hold an instance of `lsys`.

The `ca` class contains two lists of cells, the previous generation and the current one, and a pointer to the rule that allows the automata to iterate, in this case, it points to a method which implements rule 86 as discribed above (see *Approach*).

L-systems

L-systems are represented through the use of a single class, `lsys`. The `lsys` class contains a list of lists encoding the rewrite rules in the form `((symbol rewrite rewrite) (symbol2 rewrite2...) ...)` where the first element of every list is the left hand side of the rule, and the remainder of the list is a collection of symbols that the first element is rewritten to. The `lsys` class also contains a counter for determining the generation of the current string, and also a list of symbols that acts as the data string.

Composition

The `composition` class is a relatively simple way of modelling a composition. The class simply contains a title, as a way of identifing the composition, and also a list of symbols to act as the data. The symbols in this list can be any abstraction of a note at all, and in this case they are simply `JFugue` tokens.

Genetics

The genetic algorithm of the program relies on a few abstractions to accomplish it's goals. There are two genetics classes in this program, `individual` and `population`.

The `individual` class models an individual of a population, and contains the data (in this case L-systems), the fitness rating of the data, and a unique identifier.

The `population` class models a population of candidate individuals, and contains a list of those individuals, and a generation marker, to keep track of the age of the population. Also contained are percentage chances for the copy, mutate, and crossover operations to happen, as well as pointers to the mutation and crossover methods. The copy method is generic enough that any population of any individuals can use it, but the mutate and crossover functions tend to be rather specific for the application, depending on what data the individuals contain.

Description of the Genetic Algorithm

As genetic algorithm implementations vary between projects [8], a discription of how this project implements genetics is in order.

The genetic algorithm in use in this program operates on populations of individuals which consist of L-systems. The three basic genetic operators—copy, mutate, and crossover—are implemented, and act upon a population to create the next (hopefully more fit) population. This process continues until a desired average fitness for the population is reached, as determined by the fitness function chosen by the user. This process will hopefully create a population of "fit" L-systems from which to generate a melody.

Copy operator

The copy operator is fairly simple. To perform a copy, a selection of 10% of the individuals of a population is taken, and of this selection, the individual with the highest fitness rating is chosen to be copied into the next population. This is done in a direct manner—all the data from the selected individual is copied verbatim into a new individual, which is then placed into the new population.

Mutate operator

The mutate operator acts in an interesting manner. Considering that L-systems are defined in terms of rules, the mutate operator has to work on this way of structuring data. For a mutation, a favored selection (outlined above, see *Copy operator*) is performed, and the selected individual is mutated. The mutation consists of selecting one of the symbols within one of the rewrite rules at random and replacing it by a symbol chosen at random from a list of allowed symbols (the "alphabet"). This now-mutated individual is copied into the new population as outlined above.

Crossover operator

The crossover operator takes two favored selections (outlined above, see *Copy operator*), and creates a new individual out of them. To account for the fact that L-systems are rules-based systems, the crossover operator picks one rule at random from the first individual, and replaces that with it's corresponding rule from the second. This individual is then copied into the new population.

Fitness functions

The fitness functions are of great importance, as they are essentially the determining factors as to whether or not the genetic algorithm will show any true progress [8].

Two fitness functions are defined with this program, both using the same basic principle. The first is the "balancing" function. This function was designed to capture the even distribution of pitches in serialist music. This function returns a value between 0 and 1 inclusive based upon the following formula:

$$\frac{1}{\left[\frac{\left[\frac{a}{b}+\frac{b}{a}+\ldots+\frac{y}{z}+\frac{z}{y}\right]}{n\times(n-1)}\right]}, min0, max1.$$

where a, b, ... y, z are the counts of the symbols appearing in the right hand of the rewrite rules, and n is the number of symbols in the alphabet. This method works as a fairly decent way of ensuring that the symbols are equally distributed across the rewrite rules, but fails to guarantee that they will appear with equal frequency in the output of the L-system.

The second is the “balancing-output” function, designed to capture the even distribution of pitches in serialist music, however this one applies the same approach as the “balancing” fitness function to the output of the L-system, after 6 generations, in attempts to gague the system’s effectiveness at producing well-balanced output.

It should be noted that these fitness functions are far too sensitive, and muffle the results of the genetic algorithm (see *Results*).

JFugueDaemon

An auxilliary program to `ALM` that forms a part of this project is `JFugueDaemon`, a small daemon written in Java which watches for file creations in a specified directory, and upon discovering a new file reads it as input. This input is sent to `JFugue` for sonification and to save the composition as a MIDI file. `JFugue` is well suited for this purpose, as it provides an easy to use, yet fully-functional MIDI implementation, and is portable to any platform where a Java Virtual Machine is available [9].

This loose coupling removes `ALM` from the burden of playing sounds and writing MIDI and allows it the freedom of continuing along composing algorithmically without consideration of the details of MIDI generation. This sort of loose interaction between components has been lauded as one of the tenents of good algorithmic composition environment design [10].

Results

The final version of `ALM` fared fairly well. Of particular note is the performance of the fitness functions. These functions proved to be far too sensitive to unfavorable mutations in testing, oftentimes never allowing the population to come to a relatively stable level of evolution. The table following shows the fitness of the first 20 generations of a run of the genetic algorithm at 50% copy, 49% crossover, and 1% mutation:

Generation	Fitness
Initial fitness	0.28499436
1	0.5026495
2	0.6484151
3	0.6143918
4	0.36710966
5	0.64340574
6	0.7197356
7	0.602708
8	0.8284424
9	0.6633316
10	0.6908027
11	0.7634973
12	0.23890796
13	0.5888962
14	0.7547191
15	0.5993262
16	0.7726623
17	0.84089243
18	0.7832393
19	0.69761515
20	0.6627491

From this, it is clear that a single bad mutation can cause generations of good evolution to go to waste. Take, for example, the dramatic drop in fitness between generations 11 and 12.

However, once mutation was reduced to 0% (with copy and crossover both at 50%), the genetic algorithm found a rather stable level of growth:

Generation	Fitness
Initial fitness	0.26688015
1	0.47937027
2	0.63973
3	0.7401396
4	0.9233875
5	0.95210725
6	0.989678
7	0.996584
8	0.996584
And so on...	0.996584

So, without mutation, the genetic algorithm works just as expected.

The musical output of `ALM` is about what is expected for an experiement of this type. Given an alphabet containing the G-minor scale with varying durations ("`e ei f# f#i g gi a ai b bi c ci d di`"), the system was able to generate a small tune in the key of G-minor over the course of 6 iterations:

```
C GI F#I BI B F#I F# A F# F# A F# BI E BI F#I BI C A C GI F#I BI B F#I EI A AI G G F#I
B C GI B F#I BI AI B EI A AI F# A F# F#I F#I EI A AI G G C GI F#I BI B F#I BI AI B F#I
B BI E BI E F#I B BI E BI E C GI F#I BI B F#I BI AI B F#I B BI E BI E B F#I F#I B BI E
BI E B F#I C GI F#I BI B F#I F# A F# F# A F# BI E BI F#I BI C A C GI F#I BI B F#I
```

A MIDI file of this piece may be found at
`http://csc416.suspended-chord.info/final-demos/GMINOR2-6.midi`.

While not spectacularly musical, this output shows that the method used by `ALM` works.

Discussion

While the musical output of the system isn't perfect, there is still a lot to learn from this program and its implementation. The major problem with this project ended up being in the creation of reasonable fitness functions. The ones that were devised fall spectacularly short in that they exclude even the smallest amount of mutation to occur, and the output they end up giving is sub-par at best. However, for fitness metrics that know nothing about music theory, the results are in the expected vein—not terrible, but not great either. It is the author's opinion, however, that given a proper fitness function, this project could truly shine in its intended purpose.

On the programming side of things, this project was a wonderful learning experience, as it utilized several diverse technologies (Java, Common Lisp, MIDI, to name a few) and several different algorithms and AI topics and methodologies (genetic algorithms, cellular automata, L-systems, et cetera). This project also allowed for several modules to be developed which can easily be used in future algorithmic composition projects, and through this, improved the author's skills in modularizing code. All in all, this project should be considered a success, if for nothing else but the learning opportunities it afforded.

Future Work

Several different points could be brought into consideration in improving this program. First, a GUI would allow users to interact with this program much easier and faster, and with the same results. If

the GUI were written in Java, the entire project would be cross-platform, so long as `JFugue` could find a MIDI interface on the system.

Tweaking the genetic algorithm could be insightful. There are a large number of untapped possibilities in that being discarded, simply because of the parameters at play (namely the chances of copies, mutations, and crossovers).

A goal that the author fully intends to pursue is turning a large amount of the code written for this project into a Lisp module library—a library of commonly used code snippets and abstractions—written specifically with algorithmic composition in mind. Lisp is a perfect choice of a language for this, considering that it allows itself to act as a glue for connecting different modules of code together; this property is a very important consideration in designing an algorithmic composition system, as outlined by Andrea Valle [10].

Conclusion

A system for algorithmic composition isn't always the easiest thing to design [10]. Combining several different topics in AI and artificial life, `ALM` is a unique offering in the field of algorithmic composition.

The development process allowed implementation of these various algorithms, and also creation of a collection of reusable modules for future projects. The modularization of the code allows for several modifications to be made to the program readily. With a cleverly and intelligently devised fitness metric, the results of this process could be tremendously advantageous for melodic generation. Though future work is needed to push this from the realm of experiment to workable solution, a firm foundation has been laid.

To close with a quote, "Composers shouldn't think too much—it interferes with their plagiarism" [11]. While a bit blunt, this quote embodies a widely held view on composition. And if it is true, `ALM` offers a way to reduce thought and increase plagiarism, in a way that might still be considered integral.

References

[1] Gerhard Nierhaus, *Algorithmic Composition: Paradigms of Automated Music Generation*, Springer, New York, 2009.

[2] Jaime Serquera and Eduardo R. Miranda, *Algorithmic Sound Composition Using Coupled Cellular Automata*, Interdisciplinary Centre for Computer Music Research, 2010.

[3] Martin Supper, *A Few Remarks on Algorithmic Composition*, Computer Music Journal, 25(1), 2001.

[4] Jacob M. Peck, *LCompose: An L-System Approach to Generating Music*, QUEST Symposium, 2011.

[5] Bruce L. Jacob, *Composing with Genetic Algorithms*, International Computer Music Conference, 1995.

[6] John A. Biles, *GenJam: A Genetic Algorithm for Generating Jazz Solos*, International Computer Music Conference, 1994.

[7] Stephen Wolfram, *Random sequence generation by cellular automata*, Advances in Applied Mathematics, 7(2), 1986.

[8] Melanie Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1998.

[9] David Koelle, *The Complete Guide to JFugue: Programming Music in Java*, First ed., Web, 2008.

[10] Amdrea Valle, *Integrated Algorithmic Composition: Fluid systems for including notation in music composition cycle*, NIME, 2008.

[11] Howard Dietz, <http://www.brainyquote.com/quotes/quotes/h/howarddiet111816.html>, BrainyQuote.com, Xplore Inc, 2011.