L-SYSTEMS, SCORES, AND EVOLUTIONARY TECHNIQUES

Bruno F. Lourenço, José C. L. Ralha, Márcio C. P. Brandão
Departamento de Ciência da Computação, Universidade de Brasília
{brunofigueira,brandao,ralha}@cic.unb.br

ABSTRACT

Although musical interpretation of L-Systems has not been explored as extensively as the graphical interpretation, there are many ways of creating interesting musical scores from strings generated by L-Systems. In this article we present some thoughts on this subject and propose the use of genetic operators with L-System to increase variability.

1 INTRODUCTION

L-systems are powerful tools for creating models of plants and other structures that have some degree of self-similarity. Typically, an L-system consists of a set of symbols and a set of *rewritting rules* which can be applied in a parallel basis. Figure 1 shows an L-system for the famous *dragon curve* using the syntax adopted by [1].

```
#level 11
#delta 90
#axiom FX

X -> X+YF+;
Y -> -FX-Y;
-----------------------------------
         Resulting string
0 FX
1 FX+YF+
2 FX+YF++-FX-YF+
3 FX+YF++-FX-YF++-FX+YF+--FX-YF+
```

Figure 1: L-system grammar for the dragon curve.

The usual way of extracting something interesting from strings generated by L-Systems, is to interpret each symbol as a command to a imaginary turtle, in a LOGO-like manner. For such approaches, "F" means draw a segment with length d, "+" means turn the turtle $+\delta$ degrees, "-" means turn the turtle $-\delta$ degrees. "X" and "Y" are just auxiliary symbols and do not have a graphical interpretation. After a few iterations, 11 to be precise, we derive from the L-System in Figure 1 the picture shown in Figure 2. Description of more complex graphical structures and a complete overview of different types of L-Systems can be found in [1].

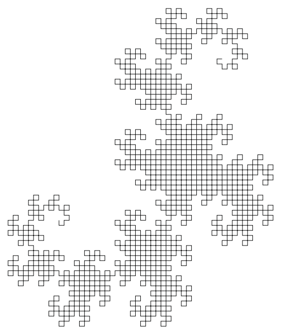


Figure 2: The dragon curve after 11 iterations on the L-system shown in Figure 1.

Of course, the graphical interpretation is not the only way to interpret the strings. Although most extensions to L-Systems focus only on the graphical interpretation, several authors described techniques to extract musical scores from strings produced by L-Systems [2], [3], [4], [5], [6]. Music has a certain fractal property [7], so it fits nicely in the context of parallel rewriting that the L-Systems provide. However, it's not a perfect fit. If the L-Systems or the rendering techniques are too simplistic, the resulting score will probably be equally simplistic, with the same theme or motif going over and over again, but starting at different points of the chosen musical scale.

In our research, we observed that the authors usually try to cope with this problem in two different ways: using more sophisticated L-Systems (stochastic or context-sensitive, for example) or using a more refined method for score generation in order to introduce variability. Keeping this idea in mind and borrowing a few operators from genetic algorithms, we have developed a method to increase variability by changing the set of rules after each iteration.

Each method of score generation has properties that make it more suitable to a certain type of L-System, but it's interesting to observe how the same L-System "behaves" under different renderings, so we have developed a program that implements three types of rendering: *spatial* [2], *sequential*

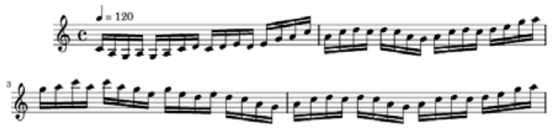


Figure 3: Score generated by projecting the graphical interpretation on a pentatonic scale

[3] and *schenkerian* [3].

Section 2 presents an overview of existing musical rendering. Section 3 introduces the notion of *Genetic L-Systems*. Section 4 describes a program written in the Python programming language that implements some musical renderings and our approach for Genetic L-Systems. Section 5 summarizes this work and suggests future directions.

2 MUSIC FROM L-SYSTEMS?

Arguably, one of the firsts articles on the subject of score generation and L-Systems was Prusinkiewicz's *Score generation with L-systems* [2], where he described a technique to extract music from the graphical interpretation of a string produced by an L-System. Each horizontal segment of the picture is interpreted as a note with a length proportional to the length of the segment. The pitch of a note is the y-th note of the chosen musical scale, where y is y-coordinate of the the segment. Figure 2 shows the first four bars of the score associated with the 9th iteration of the L-System in Figure 1.

Altough it is possible to generate interesting melodies with this *spatial rendering*, the musical rendering is still tied to the graphical rendering. So it's natural that other authors have sought to separate them. The sequential and schenkerian rendering described in [3] are examples of musical renderings that are completely independent of the graphical rendering. Both are well-suited to L-Systems that represent trees and other branched structures. The author also remarked that *"there seems to be enough information in a typical L-System to create only a short melody and still be interesting"*. To cope with this problem, Worth and Stepney suggested the use of context-sensitive and stochastic L-Systems, but some of the L-Systems built specially for the musical rendering did not have an interesting graphical interpretation, thus suggesting that it's very hard to conciliate, with aesthetic results, both the musical and the graphical interpretation.

Also worth mentioning is the LMUSe [8] program that uses map files to describe how to derive pitch, timbre and velocity from the turtle state. It has an interesting feature: a "mutate" button that randomly modifies the production rules before the first derivation step.

All the four musical renderings discussed so far are typically used with L-Systems that have sets of symbols that were originally designed for the graphical interpretation. A departure from this is the work of Jon Mccormack [4] [9], where he describes L-Systems that use notes (*A,B,C..,G*) instead of LOGO style commands (*F,+,-*).

As we have stated earlier, there is a problem with repetition and this article we try to address this issue. Even if we use stochastic rules, the set of rules is fixed, so we are still prone to hear the same fragments over and over again. The use of context-sensitive rules is a potential solution but it adds complexity to the process of building an L-System that makes sense melodically. So we want a mechanism that is simple while adding variability.

3 GENETIC L-SYSTEMS

Many authors have described techniques to "breed" L-Systems with genetic algorithms as a way of partially solving the so called *inference problem* [1] and, as a consequence, finding an L-System that produces a certain *graphical* structure. These approaches typically use genetic operators such as mutation and crossover to create new individuals (sets of production rules) and fitness functions to check if the population has a particular feature and to select the fittest individuals.

Jacob [10] described a technique to select L-Systems that produce plants with a certain branching pattern. Ashlock [11] bred populations of L-Systems to generate graphical renderings of landscapes. Mccormack [12] described an *aesthetical evolution*, where the user is asked interatively to inform the fitness of a graphical interpretation associated with a certain L-System at each step of the algorithm.

Most applications of L-Systems and evolutionary techniques are targeted to the graphical rendering. This is a surprising fact, since evolutionary techniques have been largely applied to computer music with interesting results. In [13] there is an overview of modern techniques for applying evolutionary concepts to sound synthesis and algorithmic composition. While we do not claim that we are filling the gap between the use of L-Systems and evolutionary techniques to create music, we believe that, at least, we are providing some inital steps.

There are many different techniques to mutate and to do the crossover of productions rules, which are not the hardest parts of combining genetic algorithms with L-Systems. Arguably, it's how to define the fitness function that causes the difficulties. We have taken a different aproach and decided not to use a fitness function at all. Instead, we designed an extension that allow mutations and crossover between sucessive iterations of an L-System. Consider the L-System shown on Figure 4.

[1] The inference problem asks for an axiom and set of production rules that capture a certain growth process.

```
#axiom FX

X -> X+YF+
Y -> -FX-Y,crossover(0,1)
--------------------------------------
         Resulting string
0 FX
1 FX+YF+
2 FX+YF++-FX-YF+
3 FXF++-+YFX-YF++-FX+--F+YFX-YF+
```

Figure 4: Genetic dragon curve

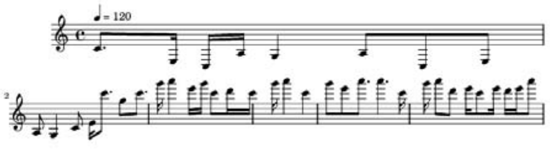


Figure 5: Spatial Rendering and a "genetic" dragon curve.

It is an extreme example, where crossover between the rules 0 and 1 is done each time the symbol `Y` is rewritten. But as odd as it may seem, comparing the score generated by spatial rendering with the original dragon curve shows a vast improvement, see Figures 3 and 5.

With parametric rules it's possible to mutate or crossover an L-System when certain conditions are met, thus allowing greater control, as shown in Figure 6.

After a rule is matched, we substitute the sucessor and then we proceed to evaluate the operators, if any. We take a simplistic view on the genetic operators as we are considering that they can only change the sucessor of the rules. More sophisticated descriptions of mutation and crossover between L-System rules can be found on [9], [10], [14], [15]. Our focus here is on the genetic operators, which are now intrinsic to the L-Systems, and that they can be parameters of the model and not just external agents. Now that's clear how we intend to use the genetic operators, we can discuss exactly how to mutate and/or do the crossover between rules.

3.1 Mutation of rules

The mutation can be thought as a function or a procedure that has two parameters: the number of the rule that will be mutated and the probability of mutation. The mutation operator scans each symbol of the sucessor of the chosen rule and then generates a random number between 0 and 1. If the generated number is less than the probability of mutation, it chooses randomly between the symbols in the

```
#axiom -XA(0)B(0)
#mutation_pool  F + -
#mutation_ignore  ( ) [ ]

X -> -YF+XFX+FY-
Y -> +XF-YFY-FX+
A(t) -> A(t+1),(t%2)==0:crossover(0,1)
A(t) -> A(t+1),(t%2)!=0:
B(t) -> B(t+1),t==3:mutation(0,0.5)
B(t) -> B(t+1),t!=3:
--------------------------------------
         Resulting string
0 -XA(0)B(0)
1 --YF+XFX+FY-A(1)B(1)
2 --+XF-YFY-F+XFX+F+-YFX+FY-F-YFX+FY-+F+
  XF-YFY-F+XFX+-A(2)B(2)
```

Figure 6: Genetic Hilbert Curve

```
#axiom -X

X -> -YF+XFX+FY-:
Y -> +XF-YFY-FX+:
--------------------------------------
         Resulting string
0 -X
1 --YF+XFX+FY-
2 --+XF-YFY-FX+F+-YF+XFX+FY-F-YF+XFX+FY-
  +F+XF-YFY-FX+-
```

Figure 7: Canonical Hilbert Curve

mutation pool and mutate the symbol in the sucessor. For example, *mutation(0,0.5)* in Figure 6 refers to a mutation in rule 0 with probability of 0.5 for each symbol.

There are certain symbols that should not be mutated because they would affect the consistency of the rules. Therefore we have to keep a list of ignored symbols, that must be skipped when we are scanning through the successor of a certain rule. In bracketed L-Systems, for example, the '[' pushes the turtle state on a stack and ']' pops the turtle state. As we usually have the same number of '[' and ']', we can not allow disruptions in this balance.

3.2 Crossover between rules

The crossover operator introduces variability by recombining two rules. Usually, we generate a random integer and then we split and combine both rules at that point. But since the sucessor of the rules can have different sizes, we have adopted the *two-point-crossover*. We generate two pseudo-random numbers for each rule, and then we swap the characters whose indexes are between these two numbers. Suppose we have generated the numbers 2,3 and 0,2 and that we have the following two rules:

```
X -> X+YF+
Y -> -FX-Y
```

The substring, in rule 0, delimited by the indexes 2 and 3 is `YF`, and the substring, in rule 1, delimited by the indexes 0 and 2 is `-FX`, so we swap them to get new rules:

```
X -> X+-FX+
Y -> YF-Y
```

If we allow bracketed or parametric rules, extra care must be taken to avoid creating unbalanced or syntatic incorrect rules. So before doing the crossover, we break the rule in tokens. Consider these two rules:

```
X -> F-[[X]+X]+F[+FX]-X
F -> FF+FF
```

We break the the first rule in F, -, [[X]+X], +, F, [+FX], - and X. The second rules gives us F,F,+,F,F. Suppose we have generated the numbers 1, 5, and 1,3. This would give us the following rules:

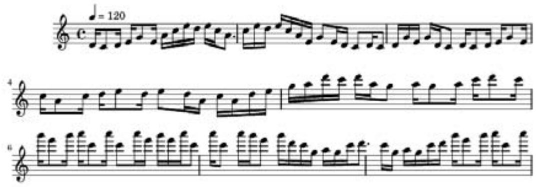
```
X -> FF+F-X
F -> F-[[X]+X]+F[+FX]F
```

3.3 Further Remarks

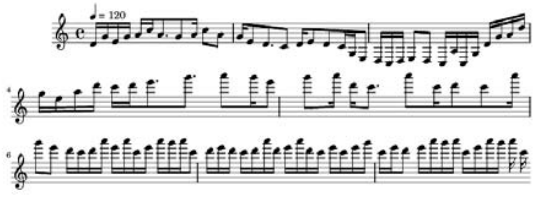
We do not describe our technique as an application of genetic algorithms because we do not have a fitness function nor a population. We pick up instead a single L-System, and we introduce the hability of mutating and to do crossover between successive iterations. Since we do not use a fitness function, the only way of evaluating the “fitness” of the musical interpretation of a certain L-System is by listening to the resulting score. In chapter 2 of *Evolutionary Computer Music* [13], John Biles argues that the most difficult part of composing music with genetic algorithms is how to specify the fitness functions, since the notion of what is right and wrong is highly subjective when we are dealing with music. Without fitness functions we open the possibility of generating scores with non-conventional musical structures, but we do not have an objective way of evaluating if the changes made by the genetic operators were positive or negative. Since our main purpose is to introduce variability, we do not impose any constraints on the operators and let the user decide himself whether a certain music piece is fit or not for his purposes.

We propose two ways of modifying an existing L-System:

- Adding a mutation or crossover operator at the end of an existing rule. The dragon curve that was discussed previously is an example of this technique.



(a) Canonical



(b) Genetic

Figure 8: Spatial rendering of the Hilbert curve

- Adding symbols to control the genetic operators. Figure 6 shows an example of this technique and Figure 7 shows the canonical Hilbert curve. Also, compare the scores produced by both L-Systems on Figure 8.

In our experiments we found out that the first technique gave the most dramatic results, since the same operator could be applied many times at each iteration thus drastically changing the set of productions rules. The second technique could be used to introduce slight deviations in the set of production rules, thus generating a score that has some traces of the original L-System.

An interesting possibility is the use of L-Systems in a similar fashion to what Mason and Saffle described on [7]. They used the spatial rendering described by Prusinkiewicz and different graphical rotations of the same L-System to build a longer musical piece while creating the feeling of counterpoint. Instead of using different rotations, we could use several realizations of the same L-System, since each realization produces a different score due to the probabilities associated with the genetic operators.

4 LSCORE

We wrote a program in Python that implements our ideas of genetic L-Systems. It is an ongoing research where we are able to generate MIDI files using different musical renderings. It’s also a parser for DOL, 2L, stochastic, bracketed and parametric L-Systems. The data flow is described on Figure 9 . It uses Python’s reflection mechanism to parse and execute the rules as Python code, allowing the user to

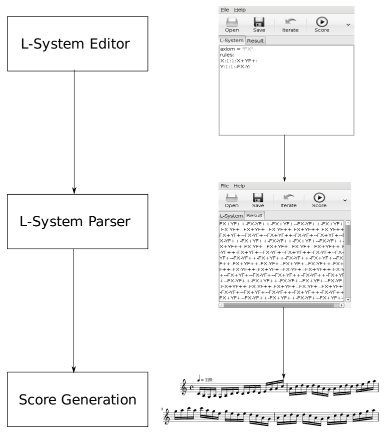


Figure 9: Data Flow in LScore

call his own routines from the production rules. The format of the rules file is shown below:

```
<Python Code>
axiom = "xy..z"
rules:
rule 1
rule 2
...
rule n
```

Each rule must use the following syntax:
predecessor:cond:prob:sucessor:code

As an example, consider the rule: `B(t):t==3:1:B(t+1):mutation(0)` The *predecessor* is `B(t)`, which is the symbol that will be replaced. The condition is `t==3`, the symbol will be replaced only if the condition is true. The probability is 1, and therefore this is a deterministic rule. The *sucessor* is `B(t+1)` and this the string that will replace the symbol `B(t)`. The *code* is `mutation(0)` and it is the procedure that will be executed after the symbol is replaced.

Figures 10 and 11, shows respectively the Genetic Hilbert and the Dragon curve written using LScore's syntax.

4.1 L-System evaluation and score generation

It is actually pretty simple to implement and evaluate L-Systems if we are dealing with non-parametric rules, since we only need to concatenate strings, check for context and generate pseudo-random numbers. But when we have parametric rules, an L-System becomes almost like a computer program of its own. Consider the following L-System:

```
axiom = "A(1)"
rules:
A(t):t==1:1:A(t*2):
```

The parametric rule `A(t):1:1:A(t*2)` matches the module `A(1)` because the letter in the production rule and in the module are the same, the number of formal parameters are also the same, and the condition (`t==1`) evaluates to true. After the rule is matched, the parameters are evaluated, the module is substituted, and we find the string `A(2)`. While it is not hard to write a parser to evaluate the arithmetic expressions that appear on parametric rules and to check if the conditions are true, a much simpler implementation is possible if we can execute statements and evaluate expressions at runtime, thus delegating the issue of expression parsing and execution to the underlying language interpreter. For that specific example, we would have two statements: `exec("t=1")` and `eval("t==1")` Since after the first statament, *t* is indeed equal to 1, the last statement returns `True` and then we can evaluate the sucessor of the rule: `eval("t*2")`, which returns 2. Both *exec* and *eval* are built-in statements in Python.

The genetic operators are coded in a similar way. As we stated earlier, each rule has a *code* part, which can be empty. We coded the genetic operators *crossover* and *mutation* as Python functions that change the production rules. After a symbol is replaced, we do `exec(code)` and the Python interpreter executes the *code* part of the rule.

After sucessive iterations, we can interpret the resulting string as a score. In our implementation, we generate a standard MIDI file with the score. The user has the option of choosing the instrument, the key, the musical scale, the method of rendering (sequential, schenkerian or spatial), the initial octave and a few other small tweaks.

The implementation of the rendering methods is straightforward, since we just have to scan the resulting string and interpret each symbol correctly according to the chosen method. At this moment, the MIDI capabilities of LScore are at best rudimentary, since we record the score in a single track and do not allow instrument changes during the rendering process. Nevertheless, it is possible to render interesting melodies and scores.

```
axiom = "-XA(0)B(0)"
rules:
X:1:1:-YF+XFX+FY-:
Y:1:1:+XF-YFY-FX+:
A(t): (t%2)==0:1:A(t+1):crossover(0,1)
A(t): (t%2)!=0:1:A(t+1):
B(t): t==3 :1:B(t+1):mutation(0)
B(t): t!=3:1:B(t+1):
```

Figure 10: Genetic Hilbert Curve with LScore's syntax. The *mutation_pool* and *mutation_ignore* are implicitely defined.

```
axiom = "FX"
rules:
X:1:1:X+YF+:
Y:1:1:-FX-Y:crossover(0,1)
```

Figure 11: Genetic Dragon Curve with LScore's syntax

5 CONCLUSION

In this article we presented a brief overview of existing methods for extracting musical scores from L-Systems and introduced a few ideas of our own. More specifically, we also presented the concept of Genetic L-Systems, where the set of productions rules can be changed between sucessive iterations. These changes are made by two genetic operators: *crossover* and *mutation*. We have also briefly described a program written in Python that implements our approach for Genetic L-Systems and generates MIDI files.

We believe we have succeeded in introducing variability in the musical interpretation of L-Systems, but certainly there is room for more experimentation. As we stated earlier, the genetic operators we are using are very simple in the sense that they only modify the sucessor of the production rules. A more sophisticated mutation process, for example, could further enhance the resulting musical score.

Also, the LScore program lacks some features, such as better MIDI support. So we have the intention of addressing these issues in the future.

6 ACKNOWLEDGEMENTS

The authors would like to thank Dr. Jon McCormack for promptly sending his PhD thesis.

7 REFERENCES

[1] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. New York: Springer-Verlag, 1990. [Online]. Available: http://algorithmicbotany.org/papers/#abop

[2] P. Prusinkiewicz, "Score generation with L-systems," *Proceedings of the 1986 International Computer Music Conference*, pp. 455–457, 1986.

[3] P. Worth and S. Stepney, "Growing music: musical interpretations of L-systems," *Springer*, vol. 3449, pp. 545–550, 2005. [Online]. Available: http://www-users.cs.york.ac.uk/~susan/bib/ss/nonstd/eurogp05.htm

[4] J. McCormack, "Grammar-based music composition." [Online]. Available: http://journal-ci.csse.monash.edu.au/ci/vol03/mccorm/mccorm.html

[5] G. L. Nelson, "Real time transformation of musical material with fractal algorithms," *Computers & Mathematics with Applications*, vol. 32, no. 1, pp. 109–116, Jul. 1996.

[6] M. Supper, "A few remarks on algorithmic composition," *Computer Music Journal*, vol. 25, no. 1, pp. 48–53, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1162/014892601300126106

[7] M. Saffle and S. Mason, "L-Systems, melodies and musical structure," *Leonardo Music Journal*, vol. 4, p. 8, 1994.

[8] D. Sharp, "Lmuse," http://www.geocities.com/athens/academy/8764/lmuse/lmuse.html, 2001.

[9] J. McCormack, "The application of L-systems and developmental models to computer art, animation, and music synthesis," Ph.D. dissertation, School of Computer Science and Software Engineering, Monash University, Clayton, Australia, 2003.

[10] C. Jacob, A. Lindenmayer, and G. Rozenberg, "Genetic L-System programming," *Parallel Problem Solving from Nature III, Lecture Notes in Computer Science*, vol. 866, pp. 334—343, 1994. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.7334

[11] D. Ashlock, S. Gent, and K. Bryden, "Evolution of l-systems for compact virtual landscape generation," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 3, 2005, pp. 2760–2767 Vol. 3.

[12] J. McCormack, "Interactive evolution of L-System grammars for computer graphics modelling," 1993. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.6763

[13] E. R. Miranda and J. A. Biles, *Evolutionary Computer Music*. Springer, April 2007.

[14] C. Roger, "On the evolution of parametric L-Systems," University of Calgary, Tech. Rep., 2000.

[15] K. Mock, "Wildwood: the evolution of L-system plants for virtual environments," in *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, 1998, pp. 476–480.