The Advantages of Generative Grammatical Encodings for Physical Design

Gregory S. Hornby
415 South Street
DEMO Lab
Brandeis University
Waltham, MA 02454
hornby@cs.brandeis.edu

Jordan B. Pollack
415 South Street
DEMO Lab
Brandeis University
Waltham, MA 02454
pollack@cs.brandeis.edu

Abstract- One of the applications of evolutionary algorithms is the automatic creation of designs. For evolutionary techniques to scale to the complexities necessary for actual engineering problems, it has been argued that generative systems, where the genotype is an algorithm for constructing the final design, should be used as the encoding. We describe a system for creating generative specifications by combining Lindenmayer systems with evolutionary algorithms and apply it to the problem of generating table designs. Designs evolved by our system reach an order of magnitude more parts than previous generative systems. Comparing it against a non-generative encoding we find that the generative system produces designs with higher fitness and is faster than the non-generative system. Finally, we demonstrate the ability of our system to go from design to manufacture by constructing evolved table designs using rapid prototyping equipment.

1 Introduction

Evolutionary algorithms (EAs) have been successfully applied to a variety of design problems [1, 2, 3], but it has yet to be shown that evolutionary techniques can scale to the complexities necessary for typical design projects. Past work has typically used a direct encoding of the solution, either by parameterizing the search space [1] or through a component based representation of the solution [4, 5, 6]. It has been argued that a generative encoding scheme, an encoding that specifies how to construct the phenotype, can achieve greater scalability through self-similar and hierarchical structure [7, 8]. In addition, by re-using parts of the genotype in the creation of the phenotype a generative encoding is a more compact encoding of a solution. Some examples of generative systems are cellular automata rules to produce 2D shapes [9], context rules to produce 2D tiles [10], graph encoding for 3D animated creatures [11], and cellular encoding for artificial neural networks [12].

Here we use Lindenmayer systems (L-systems) as a generative encoding for an EA. L-systems are a grammatical rewriting system introduced to model the biological development of multicellular organisms [13]. Rules are applied in parallel to all characters in the string just as cell divisions happen in parallel in multicellular organisms. Complex objects are created by successively replacing parts of a simple object by using the set of rewriting rules. L-systems have been used mainly to construct plants [14]. However, it is difficult to hand-make an L-system to produce a desired form. Previous work combining L-systems with EAs has been to generate plant-like structures [14, 15, 16, 17] and architectural floor designs [18] – but only limited results have been achieved.

Our work evolving L-systems uses parametric, context-free L-systems (P0L-systems), a more powerful class of L-systems than has been previously evolved. Using this system we define a component-based language for constructing objects made of voxels and define a fitness function for table designs for which we evolve table designs with thousands of voxels. We compare the generative encoding to a non-generative encoding and find that better designs evolve faster with the L-system as a generative encoding and the designs have more complex regularities than do tables created with the non-generative representation. Evolved tables are then automatically manufactured using rapid prototyping equipment.

In the following section we outline the design space and describe the components of our generative design system. We then give examples of tables evolved with the different encoding schemes and discuss the results.

2 Method

The system for creating generative designs consists of the design builder and evaluator, the L-system module and the evolutionary algorithm. L-systems are evolved by the evolutionary algorithm with individual L-systems scored for their goodness by the design builder and simulator. The end result of our system are 3D static structures; in this paper we evolve tables.

2.1 Design Builder and Evaluator

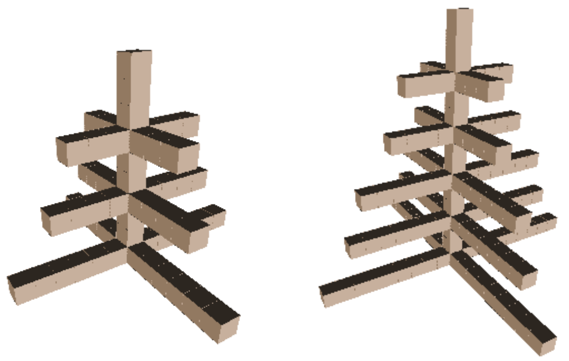
The design constructor builds a model from a sequence of build commands. Once built, a model is simulated and evaluated. Commands are listed in table 1.

The command string consists of a sequence of build commands that give instructions to a LOGO-style turtle that is used to construct an object out of voxels. The 3D matrix of voxels starts out empty and voxels are filled as the turtle enters them. [and] push and pop the current state – consisting of the current and orientation – to and from a stack. Forward moves the turtle forward in

Table 1: Design Language

Command	Description	Symbol
[ ]	push/pop orientation to stack	[ ]
{ *block* }(n)	repeat enclosed block n times	{ }
forward(n)	move in the turtle's positive X direction n units	f
backward(n)	move in the turtle's negative X direction n units	b
up(n)	rotate heading $n \times 90°$ about the turtle's Z axis	$\wedge$
down(n)	rotate heading $n \times -90°$ about the turtle's Z axis	v
left(n)	rotate heading $n \times 90°$ about the turtle's Y axis	$<$
right(n)	rotate heading $n \times -90°$ about the turtle's Y axis	$>$
clockwise(n)	rotate heading $n \times 90°$ about the turtle's X axis	/
counter-clockwise(n)	rotate heading $n \times -90°$ about the turtle's X axis	\

the current direction and backwards moves the turtle back one space, both place a block in the space if none exists. Turn left/right/up/down/clockwise/counter-clockwise rotate the turtle's heading about the appropriate axis in units of 90°. Command sequences enclosed by { } are repeated a number of times specified by the brackets' argument.



(a) (b)

Figure 1: Two example structures.

For example, the string,
[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(3)] left(1) }(4)] up(1) forward(3) down(1) forward(3)
is interpreted as:
[[forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1)] up(1) forward(3) down(1) [[forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1)] up(1) forward(3) down(1) [[forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1)] up(1) forward(3) down(1) forward(3)
and produces the structure in figure 1.a.

As the construction language only allows voxels to be placed next to existing voxels, evolved designs are guaranteed to generate a single, connected structure. The design simulator then determines the stability of the object. Once an L-system specification is executed, and the stability of the object is determined, the resulting structure is evaluated using the fitness function described in section 3.

2.2 Parametric 0L-Systems

The class of L-systems used as the encoding is a parametric, context-free L-system (P0L-system). Formally, a P0L-system is defined as an ordered quadruplet, G = (V, Σ, ω, P) where,
V is the alphabet of the system,
Σ is the set of formal parameters,
$\omega \in (V \times \Re^*)^+$ is a nonempty parametric word called the axiom, and
P $\subset (V \times \Sigma^*) \times C(\Sigma) \times (V \times \xi(\Sigma))^*$ is a finite set of productions.
The symbols : and $\rightarrow$ are used to separate the three components of a production: the predecessor, the condition and the successor. For example, a production with predecessor A(n0, n1), condition n1>5 and successor B(n1+1)cD(n1+0.5, n0-2) is written as:

$$A(n0, n1) : n1 > 5 \rightarrow B(n1+1)cD(n1+0.5, n0-2)$$

A production matches a module in a parametric word iff the letter in the module and the letter in the production predecessor are the same, the number of actual parameters in the module is equal to the number of formal parameters in the production predecessor, and the condition evaluates to true if the actual parameter values are substituted for the formal parameters in the production.

For implementation reasons we add constraints to our P0L-system. The condition is restricted to be comparisons as to whether a production parameter is greater than a constant value. Parameters to design commands are either a constant value or a production parameter. Parameters to productions are equations of the form: $[\ production\ parameter \mid constant\ ]\ [\ + \mid - \mid \times \mid \backslash\ ]\ [\ production\ parameter \mid constant\ ]$. The following is a P0L-system using the language defined in table 1 and consists of two productions with each production containing one condition-successor pair:

$$\begin{aligned} &P0(n0): \\ &n0 > 1.0 \rightarrow \quad [\ P1(n0 * 1.5)\ ]\ up(1)\ forward(3) \\ &\qquad\qquad\qquad down(1)\ P0(n0-1) \end{aligned}$$

$$\begin{aligned} &P1(n0): \\ &n0 > 1.0 \rightarrow \quad \{\ [\ forward(n0)\ ]\ left(1)\ \}(4) \end{aligned}$$

Starting this P0L-system with P0(4), produces the following sequence of strings,

P0(4)

[P1(6)] up(1) forward(3) down(1) P0(3)

[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [P1(4.5)] up(1) forward(3) down(1) P0(2)

[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [P1(3)] up(1) forward(3) down(1) P0(1)

[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(3)] left(1) }(4)] up(1) forward(3) down(1)

The last string of commands produces the tree in figure 1.a. Trees of arbitrary size can be created by starting the production system with a different argument – the tree in figure 1.b is created from this system by starting it with P0(6).

2.3 Evolutionary Algorithm

An evolutionary algorithm is used to evolve individual L-systems. The initial population of L-systems is created at random and then evolution then proceeds by iteratively selecting a collection of individuals with high fitness for parents and using them to create a new population of individual L-systems through mutation and recombination. In addition to the L-system, each individual also contains values for the initial calling parameters of the first production rule and the maximum number of iteration updates to be performed. We now describe how the initial population of L-systems are generated and then how variation is applied to them.

2.4 Initialization

L-systems have a predetermined number of production rules with a fixed number of arguments and production bodies. A new L-system is created by generating a random string of 3 to 8 build commands for each production body – for trials using a single command string this 4 to 104 commands – in blocks of 1 to 3 commands. Each added block can be enclosed by push/pop brackets *[a(1)]*, block-repetition parenthesis { *b(1) c(2)* }*(3)*, or not at all *d(1) e(2) f(3)*.

2.5 Mutation

Mutation creates a new individual by copying the parent individual and making a small change to it. First a production rule is selected at random from one of the used production rules and then this rule is changed in some way. Changes that can occur are: replacing one command with another; perturbing the parameter to a command by adding/subtracting a small value to it; changing the parameter equation to a production; adding/deleting a sequence of commands in a successor; changing the condition equation; or encapsulating a block of commands and turning it into a, previously unused, production rule.

For example, if the production $P0$ is selected to be mutated,

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

some of the possible mutations are,
Mutate the condition:

$$\begin{array}{ll} P0(n0,n1): & n0 > \mathbf{5.0} \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

Mutate an argument:

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(\mathbf{n1-2.0}, n0/2.0)\ ] \end{array}$$

Mutate a symbol:

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ \mathbf{c(1.0)}\ b(2.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

Delete random character(s):

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ a(1.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

Insert a random sequence of character(s):

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1)\ \mathbf{c(3.0)} \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

Encapsulate a block of characters:

$$\begin{array}{ll} P0(n0,n1): & n0 > 5.0 \rightarrow \{\ \mathbf{P2(n0,n1)}\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \\ \mathbf{P2(n0,n1)}: & \mathbf{n0 > 5.0 \rightarrow a(1.0)\ b(2.0)} \\ & \mathbf{n0 > 2.0 \rightarrow a(1.0)\ b(2.0)} \end{array}$$

2.6 Recombination

Recombination takes two individuals, $p1$ and $p2$, as parents and creates one child individual, c, by making it a copy of $p1$ and then inserting a small part of $p2$ into it. This is done by replacing one successor of c with a successor of $p2$, inserting a sub-sequence of commands from a successor in $p2$ into c, or replacing a sub-sequence of commands in a successor of c with a sub-sequence of commands from a successor in $p2$.

For example if parent 1 has the following rule,

$$\begin{array}{ll} P3(n0,n1): & n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ & n0 > 2.0 \rightarrow [\ P1(n1-1.0, n0/2.0)\ ] \end{array}$$

and parent 2 has the following rule,

$$\begin{array}{ll} P3(n0,n1): & n1 > 3.0 \rightarrow b(3.0)\ a(2.0) \\ & n0 > 1.0 \rightarrow P1(n1-1.0, n1-2.0) \end{array}$$

Then some of the possible results of a recombination on successor P3 are:
Replace an entire condition-successor pair:

$$P3(n0, n1): \quad \begin{array}{l} \mathbf{n1 > 3.0 \rightarrow b(3.0)\ a(2.0)} \\ n0 > 2.0 \rightarrow [\ P1(n1 - 1.0, n0/2.0)\ ] \end{array}$$

Replace just a successor:

$$P3(n0, n1): \quad \begin{array}{l} n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ n0 > 2.0 \rightarrow \mathbf{P1(n1 - 1.0, n1 - 2.0)} \end{array}$$

Replace one block with another:

$$P3(n0, n1): \quad \begin{array}{l} n0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n1) \\ n0 > 2.0 \rightarrow [\ \mathbf{b(3.0)\ a(2.0)}\ ] \end{array}$$

3 Experiments and Results

In this section we present the results of evolving tables with the non-generative encoding, and the P0L-system as a generative encoding. All trials are run for a maximum of 1000 generations using an evolutionary algorithm with 100 individuals. The best 2 individuals of one generation are copied to the next (an elitism of 2) and remaining individuals are created with an equal probability of using mutation or recombination. The grid size for evolved tables is 40 wide $\times$ 40 deep $\times$ 40 high – except for the the grid used for tables that were manufactured, which use a grid of 50 wide $\times$ 20 deep $\times$ 20 high.

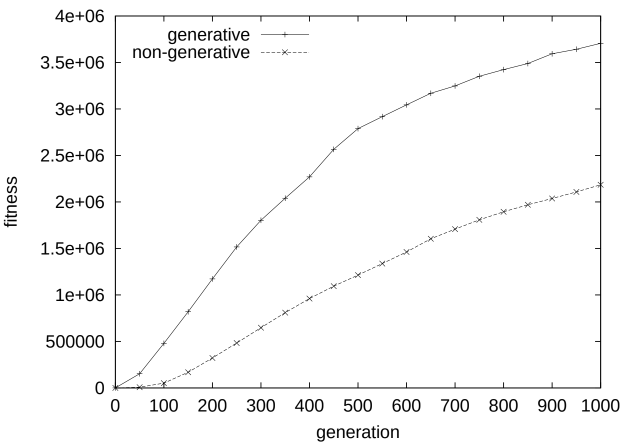


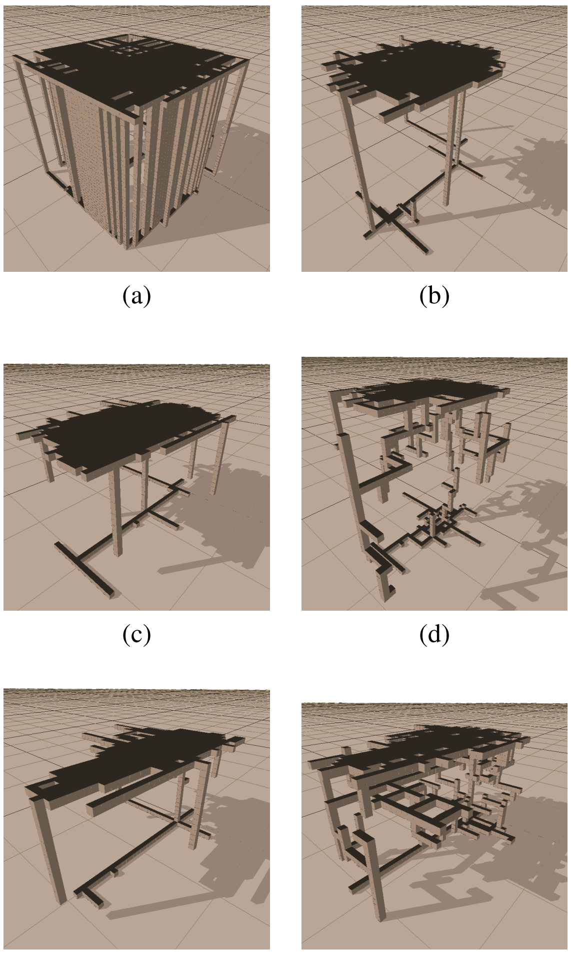
Figure 2: Performance comparison between the non-generative encoding and the P0L-system generative encoding.

The fitness of a table is a function of its height, surface structure, stability and number of excess voxels used. Height is the number of voxels above the ground. Surface structure is the number of voxels at the maximum height. Stability is a function of the volume of the table and is calculated by summing the area at each layer of the table, except for the surface. Maximizing height, surface structure and stability typically result in table designs that are solid volumes, thus a measure of excess voxels is used to reward designs that use fewer bricks.

$$\begin{aligned} f_{height} &= \text{the height of the highest voxel, } Y_{max}. \\ f_{surface} &= \text{the number of voxels at } Y_{max}. \\ f_{stability} &= \sum_{y=0}^{Y_{max}-1} f_{area}(y) \\ f_{area}(y) &= \text{area in the convex hull at height y.} \\ f_{excess} &= \text{number of voxels not on the surface.} \end{aligned}$$

For these experiments we combine these measures into a single function [1],

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{excess} \tag{1}$$



(a) (b) (c) (d) (e) (f)

Figure 3: Tables evolved using a non-generative encoding.

[1] A more appropriate method of evolving against these criteria may be to use a multi-objective approach [19].

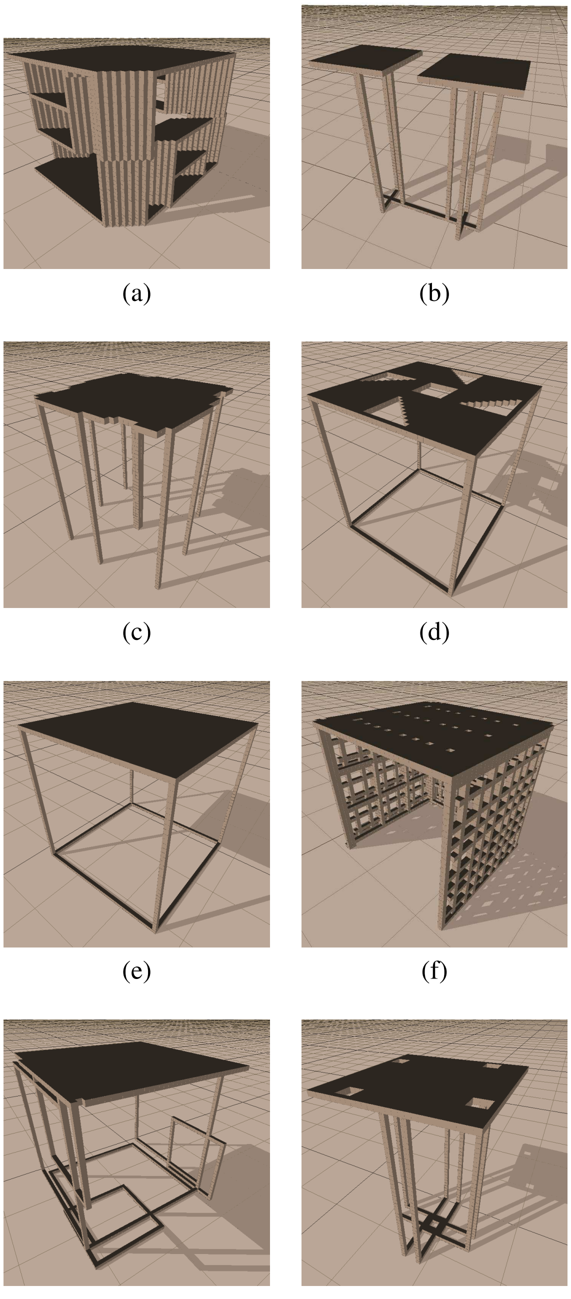


Figure 4: Tables evolved using the P0L-system as a generative encoding.

To determine if generative encodings are better than non-generative encodings we ran two sets of experiments using equation 1 for the fitness function. For a non-generative encoding, the command language listed in table 1 was used without the block replication operator. Each individual consisted of a single sequence of a maximum of 20000 commands. For the generative encoding we used the P0L-system described in section 2.2 with 20 production rules, a maximum of 3 condition-successor pairs and production rules had 2 parameters and the command language of table 1. Both encodings used the same mutation and recombination operators. The graph in figure 2 plots the average, maximum fitness over 100 trials for the two different encodings and shows that the average fitness using the P0L-system as a generative encoding is significantly better than the non-generative encoding.

Different weightings and variations of the components of the fitness function were tried but in all cases the generative encoding produced better tables than the non-generative encoding. Table designs for both systems tended to have from 1500 to 2500 voxels, the largest is shown in figure 4.a and consists of 5921 voxels. Examples of high-fitness designs evolved with the non-generative encoding are shown in figure 3. On evolutionary runs with this system over half the trials converged to poor structures and no run produced structures with complex regularities. Tables evolved using the generative encoding encoding are shown in figure 4 and figure 6. Most trials converged to good structures, and even those with low fitness had regularities.

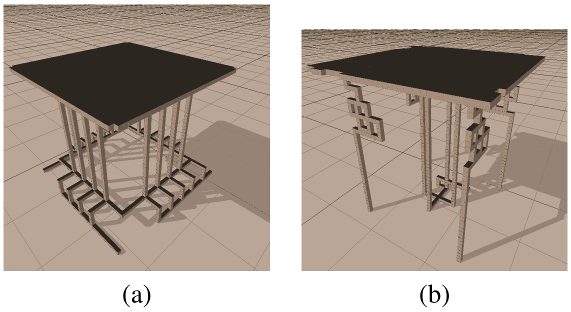
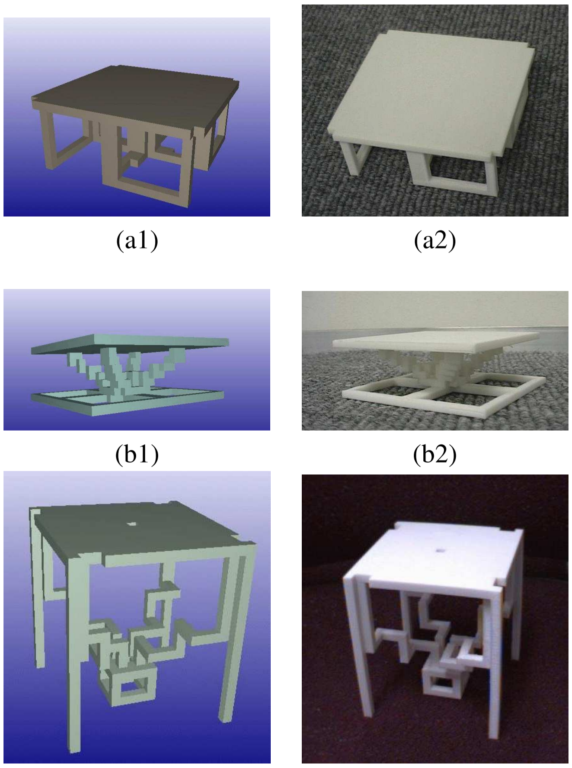


Figure 5: Tables evolved using: a, a non-L-system, generative encoding with block replication; and b, a P0L-system encoding without block replication.

The difference between the non-generative encoding and the generative, P0L-system encoding are block replication and parametric production rules. Using only one of block replication or parametric production rules there are two variants of the generative encoding, with which we performed a small number of evolutionary runs. The first variant consisted of removing production rules from the generative system and leaving block replication. Also, the single command sequence of this encoding was allowed to grow to 20000 commands. This system was not as good as the original generative encoding with production rules, but it produced better tables than did the non-generative encoding. The other variant we tried was the P0L-system without block replication. This system also performed worse than the original P0L-system with block-replication but better than the non-generative encoding. Both variants produced tables with regularities, although the second variant did not necessarily have the sequential replications of structure as did the first variant, and both are improvements over the non-generative encoding. Figure 5 contains a table evolved with both alternative, generative encodings.

Automated manufacture of evolved table designs is achieved by use of rapid-prototyping equipment. Designs that



(a1) (a2) (b1) (b2) (c1) (c2)

Figure 6: Manufacture tables shown both in simulation (left) and reality (right).

are generated by the evolutionary system are saved to a file format describing their shape. Tables are then constructed by printing them on a 3D printer, shown in figure 6.

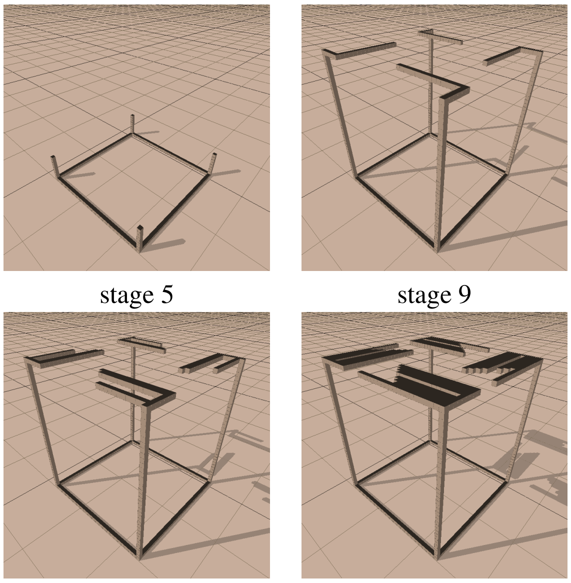
4 Discussion

Regardless of how it is achieved, a generative encoding should incorporate a bias towards re-used modules. Re-using code to re-use parts in the actual design makes certain types of design changes easier. For example, if a table is created from modules then changing the length of each table-leg requires only one change in the leg-building module. A direct encoding scheme would require this change to be made at each occurrence of a table-leg in the genotype. As designs become more complex, the likelihood of the same change happening simultaneously to all uses of a part becomes increasingly unlikely in a direct encoding, but does not change in a modular, generative encoding.

The following is the generative encoding for the table in figure 4.d, and has a genotypic structure that is related to the structure of the table,

```
P0:
 (n1>3.0) :- P11(n0/4.0,n0=2.0) v(1.0)
             {P17(n1=3.0,n1/2.0) P18(n1+n0,n0+n1)
             P3(n1=1.0,n1-n0) }(4.0)

P2:
 (n0>0.0) :- [<(4.0) P12(n1=3.0,n0=4.0) ]

P3:
 (n1>2.0) :- P16(n1=4.0,n0-n1) P16(n1=4.0,n0-n1)
             P16(n1=4.0,n0-2.0) P16(n1=4.0,n0-n1)

P6:
 (n1>1.0) :- [b(5.0) <(1.0) b(5.0) v(1.0) ^(1.0)
             b(5.0) b(5.0) b(5.0) ]
 (n0>1.0) :- [b(5.0) ^(1.0) b(n0) <(1.0) b(n0)
             v(5.0) ^(1.0) b(5.0) <(n1) b(5.0)
             v(1.0) ]

P7:
 (n0>-1.0) :- [/(1.0) /(1.0) <(1.0) /(1.0) /(5.0)
              >(1.0) /(1.0) ] v(1.0)

P8:
 (n0>0.0) :- P8(n0/4.0,n1+1.0) [b(4.0) b(4.0)
             P8(n1-2.0,n0-5.0) ]
 (n1>-2.0) :- [P8(n0/4.0,n1+1.0) b(5.0)
              P8(n1-5.0,n0-5.0) b(4.0) b(4.0)
              b(4.0) P6(n1-n0,n0+n1) ]

P9:
 (n1>3.0) :- P7(3.0-3.0,n0+n1) /(1.0)
             P8(n1-n0,n1+1.0)

P11:
 (n1>-10.0) :- >(1.0)

P12:
 (n1>0.0) :- \(4.0)

P14:
 (n0>10.0) :- P2(n1/3.0,n1+n0) P9(n0=n1,n0/n1)

P16:
 (n1>22.0) :-
 (n1>5.0) :-
 (n1>0.0) :- [P19(n0/2.0,n1-n0) b(1.0) f(3.0) ]
             /(2.0) P3(n0=5.0,n1-5.0)

P17:
 (n1>3.0) :- ^(n1) b(2.0) b(4.0) b(n0) b(3.0)
             b(3.0) b(5.0)

P18:
 (n1>3.0) :- b(n1) P14(n1-3.0,n1=3.0) <(1.0)
             <(1.0) >(1.0) >(1.0)
```

This L-system is started with the command $P0(4.0, 10.0)$ and goes through 17 iterations of parallel replacement – structures created by some of these intermediate stages are shown in figure 7. The first iteration produces the string, *P11(1,2) v(1) {P17(3,5) P18(14,14) P3(1,6) }(4)*, which uses block replication to encode that the table has four legs – reducing the block-replication parameter to 3 results in a 3-legged table. Within this block, the productions $P17$, $P18$ and $P3$ are called once, and this is the only time they are called. The structure of the base is encoded in productions $P17$ and $P18$. Reducing the number of voxels created from its $backward()$ command, $b()$, in these productions reduces the width and depth of the table. $P3$ calls $P16$, but all of $P16$'s conditions fail, and this sequence of productions produces no build commands. From $P18$ there are calls to $P14$ then $P9$ – $P14$



stage 5 stage 9

stage 10 stage 12

Figure 7: Growth of a table.

also calls $P2$ which then calls $P12$, but none of these productions produce bricks. $P9$ changes the direction of the turtle to build the table legs with the help of $P7$, and then the table legs and surface are encoded in productions $P6$ and $P8$, which construct the legs and surface through repeated calls to each other. In both $P6$ and $P8$ the parameter values are used to select which production body to use. The height of the legs is encoded in the first successor, for which the condition succeeds when $P8$ is initially called and then for the first time it calls itself. In later calls to $P8$ the first condition fails and the second condition succeeds resulting in the first call to $P6$ and this begins the sequence of commands for constructing the table's surface. Later evolution changed the production rules $P6$ and $P8$ to,

```
P6:
   (n1>1.0) :- [b(5.0) ^(5.0) b(n0) <(1.0) b(5.0)
               b(5.0) b(5.0) b(4.0) ]
   (n0>2.0) :- [b(5.0) ^(5.0) b(n0) b(5.0) <(1.0)
               v(5.0) ^(5.0) b(5.0) b(5.0) b(5.0)
               b(4.0) ]
   (n1>0.0) :- [b(5.0) ^(5.0) <(1.0) b(n0) b(5.0)
               v(5.0) ^(1.0) b(5.0) b(5.0) b(5.0)
               b(4.0) ]

P8:
   (n0>0.0) :- P8(n0/5.0,n1+1.0) [b(4.0) b(4.0)
               P8(n1-2.0,n0-5.0) ]
   (n1>-2.0) :- [P8(n0/4.0,n1+1.0) b(5.0) b(4.0)
               P8(2.0-5.0,3.0-5.0) b(4.0) b(5.0)
               P6(n1-n0,n0+n1) ]
   (n0>-1.0) :- \(1.0) v(3.0) v(n0)
```

with the resulting table shown in figure 4.e. By counting each production head as 1 character, each condition as 2 characters and 1 for each character in the production body the specification length of the encoding for the table in figure 4.d is 123 characters and encodes into a command sequence of 5574 commands – a compression factor of over 45 – to produce a table of 1280 voxels.

Block replication and production rules with parameters differentiate this work from previous work in evolving L-systems [15, 20, 17, 18]. Both block replication and production rules are similar to features of past work in evolutionary design and both have analogues in computer languages. Block replication is similar to $for\text{-}next$ loops in computer programs and is almost identical to the multiple re-writing of the recurrent symbol (using the life register) of cellular encoding [21] and the recursive-limit parameter in graph encoding [22]. Production rules are like subroutine calls in programming languages and are similar to the automatically defined sub networks (ADSNs) of cellular encoding and automatically defined functions (ADFs) of genetic programming [23]. With analogues to loops and parameterized subroutines, evolution of generative encodings becomes like the evolution of a computer program, as in genetic programming [23].

A beauty of L-systems as a generative encoding is that it is a general, generative encoding system. By changing the language of terminals different structures can be generated, such as plants [14], artificial neural networks [24], and locomoting creatures [25, 26].

5 Conclusion

A system for automatically producing generative design systems with regular structure was achieved by using parametric Lindenmayer-systems as the generative encoding for an evolutionary algorithm. To compare performance between the generative encoding and a non-generative encoding we defined a voxel-based language for building structures and a fitness function for evaluating table designs constructed by this language. Using this system, tables with thousands of voxels were evolved, an order of magnitude more parts than previous generative systems [9, 10]. Evolution using the L-system as a generative encoding was both more consistent at producing good table designs and produced better results faster than the non-generative, component-based encoding.

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant No. DASG60-99-1-0004. The authors would like to thank the members of the DEMO Lab: A. Bucci, E. DeJong, S. Ficici, P. Funes, S. Levy, H. Lipson, O. Melnik, S. Viswanathan and R. Watson.

Bibliography

[1] P. Husbands, G. Germy, M. McIlhagga, and R. Ives. Two applications of genetic algorithms to component design. In T. Fogarty, editor, *Evolutionary Computing. LNCS 1143*, pages 50–61. Springer-Verlag, 1996.

[2] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

[3] P. J. Bentley, editor. *Evolutionary Design by Computers*. Morgan Kaufman, 1999.

[4] Couro Kane and Marc Schoenauer. Genetic operators for two-dimentional shape optimization. In J.-M. Alliot, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificiale Evolution - EA95*. Springer-Verlag, 1995.

[5] P. J. Bentley. *Generic Evolutionary Design of Solid Objects Using a Genetic Algorithm*. PhD thesis, University of Huddersfield, 1996.

[6] P. Funes and J. Pollack. Computer evolution of buildable objects. In Phil Husbands and Inman Harvey, editors, *Proceedings of the Fourth European Conference on Artificial Life*, pages 358–367, Cambridge, MA, 1997. MIT Press.

[7] Marc Schoenauer. Shape representations and evolution schemes. In L. J. Fogel, P. J. Angeline, and T. Bäck, editors, *Evolutionary Programming 5*. MIT Press, 1996.

[8] P. J. Bentley. Exploring component-based representations - the secret of creativity by evolution? In *Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM 2000)*, 2000.

[9] Hugo de Garis. Artificial embryology : The genetic programming of an artificial embryo. In Branko Soucek and the IRIS Group, editors, *Dynamic, Genetic and Chaotic Programming*. Wiley, 1992.

[10] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem. In Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiel, and Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 35–43, 1999.

[11] Karl Sims. Evolving Virtual Creatures. In *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 15–22, 1994.

[12] Frédéric Gruau. *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Supérieure de Lyon, 1994.

[13] A. Lindenmayer. Mathematical models for cellular interaction in development. parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315, 1968.

[14] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[15] C. Jacob. Genetic L-system Programming. In Y. Davidor and P. Schwefel, editors, *Parallel Problem Solving from Nature III, Lecture Notes in Computer Science*, volume 866, pages 334–343, 1994.

[16] C. Traxler and M. Gervautz. Using genetic algorithms to improve the visual quality of fractal plants generated with csg-pl-systems. In *Proc. Fourth International Conference in Central Europe on Computer Graphics and Visualization 96*, 1996.

[17] G. Ochoa. On genetic algorithms and lindenmayer systems. In A. Eiben, T. Baeck, M. Schoenauer, and H. P. Schwefel, editors, *Parallel Problem Solving from Nature V*, pages 335–344. Springer-Verlag, 1998.

[18] P. Coates, T. Broughton, and H. Jackson. Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In P. J. Bentley, editor, *Evolutionary Design by Computers*, 1999.

[19] Carlos M. Fonseca and Peter J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16, 1995.

[20] C. Traxler and M. Gervautz. Using genetic algorithms to improve the visual quality of fractal plants generated with csg-pl-systems. In *Proc. Fourth Intl. Conf. in Central Europe on Computer Graphics and Visualization 96*, 1996.

[21] Frédéric Gruau and Kameel Quatramaran. Cellular encoding for interactive evolutionary robotics. Technical Report 425, University of Sussex, 1996.

[22] Karl Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, pages 28–39, Boston, MA, 1994. MIT Press.

[23] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, Mass., 1992.

[24] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[25] G. S. Hornby, H. Lipson, and J. B. Pollack. Evolution of generative design systems for modular physical robots. In *Intl. Conf. on Robotics and Automation*, 2001.

[26] G. S. Hornby and J. B. Pollack. Body-brain coevolution using l-systems as a generative encoding. In *Genetic and Evolutionary Computation Conference*, 2001.