Body-Brain Co-evolution Using L-systems as a Generative Encoding

Gregory S. Hornby
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
hornby@cs.brandeis.edu

Jordan B. Pollack
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
pollack@cs.brandeis.edu

Abstract

We co-evolve the morphology and controller of artificial creatures using two integrated generative processes. L-systems are used as the common generative encoding for both body and brain. Combining the languages of both into a single L-system allows for linkage between the genotype of the controller and the parts of the morphology that it controls. Creatures evolved by this system are more complex than previous work, having an order of magnitude more parts and a higher degree of regularity.

1 INTRODUCTION

As computers become more powerful the richness of virtual worlds is limited only by what can be designed to inhabit them. How can we construct beautiful and complex designs, objects and creatures for them? 3D virtual creatures have been evolved in simulation [Komosinski & Rotaru-Varga, 2000], and simple robots have been evolved for automatic manufacture [Lipson & Pollack, 2000]. Both of these works have used a direct encoding for the creature morphology and controller. In this we return to the spirit of [Sims, 1994], in which a graph structure was the generative encoding evolved for creating both the body and brain of virtual creatures.

Previously we showed that an evolutionary algorithm (EA) using a Lindenmayer system (L-system) as a generative encoding outperformed an EA using a non-generative encoding on an automated design problem [Hornby & Pollack, 2001]. We then used this system to evolve complex morphologies for 2D robots with motorized joints, each controlled by an oscillator [Hornby et al., 2001]. Here we describe extensions of this work to 3D creatures and to the integration of neural networks as controllers. Advantages of neural networks are that they can generate more complex locomotion patterns than unconnected oscillators and allow for later progression to the evolution of morphologies with sensors and reactive controllers. Integrating the commands for morphology and controller in the genotype creates a linkage between them, like the encoding of [Sims, 1994], and should reduce disruption under recombination.

L-systems have been used previously for the development of artificial neural networks. In [Kitano, 1990] an L-system on matrices was used to generate the connectivity matrix of a network. This method does not naturally extend to the co-evolution of morphology along with the neural controller. More compatible with our original system of producing a string of build commands is the technique of [Boers & Kuiper, 1992]. Here groupings of symbols inside brackets are used to specify connectivity of the network. Drawbacks to this system are that a symbol is used for each neuron, which limits its ability to scale to large networks. Our system creates networks in a method similar to that of cellular encoding [Gruau, 1994], with operators acting on links instead of on the nodes, as in [Luke & Spector, 1996].

Using this system we evolve both the neural controllers and morphologies of creatures for locomotion. Whereas the generative encoding of [Sims, 1994] allows for repetition of segments, it did not produce hierarchies of regularity. Our generative encoding system is a more powerful language with loops, sub-procedure-like elements, parameters and conditionals and achieves an order of magnitude more parts than the previous work of [Komosinski & Rotaru-Varga, 2000], [Lipson & Pollack, 2000] and [Sims, 1994].

In the following sections we first outline the design space and describe the components of our generative

design system, then we present our results and finally close with a discussion and conclusion of our work.

2 EXPERIMENTAL METHOD

In these experiments we evolve a generative genotype that specifies how to construct both the morphology and controller of a locomoting creature. A Lindenmayer system (L-system) is used as the generative specification system for both body and brain and is optimized by an evolutionary algorithm (EA). The system consists of the network and morphology constructor, the L-system parser, the evolutionary algorithm and the physics and network simulator.

2.1 MORPHOLOGY CONSTRUCTOR

The morphology constructor and simulator is a 3D extension of the 2D work in [Hornby et al., 2001]. The morphology constructor builds a model from a string of build commands to a LOGO-style turtle [Abelson & deSessa, 1982] using a command language similar to that of L-system languages for creating plants [Prusinkiewicz & Lindenmayer, 1990]. As the turtle moves, bars are created and these become the morphology of the creature. The commands instruct the turtle to move forward or backward and to change orientation, and there are commands for creating actuated joints.

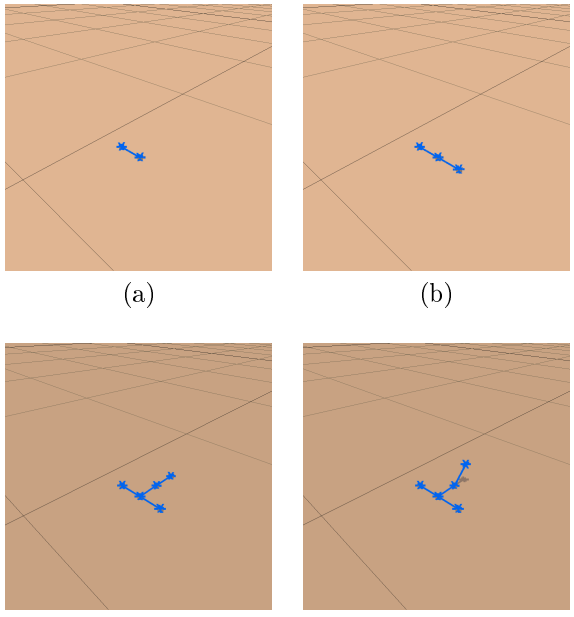


Figure 1: Building And Simulating A 3D Creature

The commands for this language are: '[' ']', store/retrieve the current state (consisting of the current location and orientation) to and from a stack; { *block* }(n), repeat the enclosed block of build commands n times; *forward*, moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar; *backward*, goes back up the parent of the current bar; *revolute-1*, forward, end with a joint with range 0° to 90° about the current Z-axis; *revolute-2*, forward, end with a joint with range −45° to 45° about the current Z-axis; *twist-90*, forward, end with a joint with range 0° to 90° about the current X-axis; *twist-180*, forward, end with a joint with range −90° to 90° about the current X-axis; *up*(n), rotate heading $n \times 90°$ about the turtle's Z axis; *down*(n), rotate heading $n \times -90°$ about the turtle's Z axis; *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; and *counter-clockwise*(n), rotate heading $n \times -90°$ about the turtle's X axis.

When evolving oscillating motors, and not neural controllers, the oscillation function of a joint is specified as in [Hornby et al., 2001] by adding a parameter to *revolute-1*, *revolute-2*, *twist-90*, and *twist-180* to specify the rate of oscillation and by adding the following two commands to control the phase off: *increase-offset*(n), increase phase offset by $n \times 25\%$ and *decrease-offset*(n), decrease phase offset by $n \times 25\%$.

An example of a creature constructed using this language is shown in figure 1. The single bar in figure 1.a is built from the string, *[left(1) forward(1)]*, and the two bar structure in figure 1.b is built from, *[left(1) forward(1)] [right(1) forward(1)]*. The final creature is made from the command sequence, *[left(1) forward(1)] [right(1) forward(1)] revolute-1(1) forward(1)*, and is shown in figure 1.c. Figure 1.d displays the creature with the actuated joint moved half-way through its joint range.

2.2 NETWORK CONSTRUCTOR

The method for constructing the neural controllers for the artificial creatures is similar to that of cellular encoding [Gruau, 1994], with two main differences. The first difference between this work and that of cellular encoding is that strings of build commands are used instead of trees of build commands, although the *push* and *pop* operators (described later) add a branching ability to the strings. The other difference is that build commands operate on the links connecting the nodes as with edge encoding [Luke & Spector, 1996] instead of on the nodes of the network. Advantages of edge

encoding are that at most one link is created with a build command so each build command can specify the weight to attach to that link and, unlike cellular encoding, sub-sequences of build commands will construct the same sub-network independent of where in the build-tree they are located.

Commands for constructing the network operate on links between neurons and use the most recently created link as the current one. *Push* and *pop* operators, '[' and ']', are used to store and retrieve the current link – consisting of the from-neuron, the to-neuron and index of the link into the to-neuron (for when there are multiple links between neurons) – to and from the stack. This stack of edges allows a form of branching to occur in an encoding – an edge can be pushed onto the stack followed by a sequence of commands and then a pop command makes the original edge the current edge again. For the following list of commands the current link connects from neuron A to neuron B.

- decrease-weight(n) – Subtracts n from the weight of the current link. If the current link is a virtual link, it creates it with weight $-n$.
- duplicate(n) – Creates a new link from neuron A to neuron B with weight n.
- increase-weight(n) – Add n to the weight of the current link. If the current link is a virtual link, it creates it with weight n.
- loop(n) – Creates a new link from neuron B to itself with weight n.
- merge(n) – Merges neuron A into neuron B by copying all inputs of A as inputs to B and replacing all occurrences of neuron A as an input with neuron B. The current link then becomes the nth input into neuron B.
- next(n) – Changes the from-neuron in the current link to its nth sibling.
- output(n) – Creates an output-neuron, with a linear transfer function, from the current from-neuron with weight n. The current-link continues to be from neuron A to neuron B.
- parent(n) – Changes the from-neuron in the current link to the nth input-neuron of the current from-neuron. Often there will not be an actual link between the new from-neuron and to-neuron, in which case a virtual link of weight 0 is used.
- reverse – Deletes the current link and replaces it with a link from B to A with the same weight as the original.

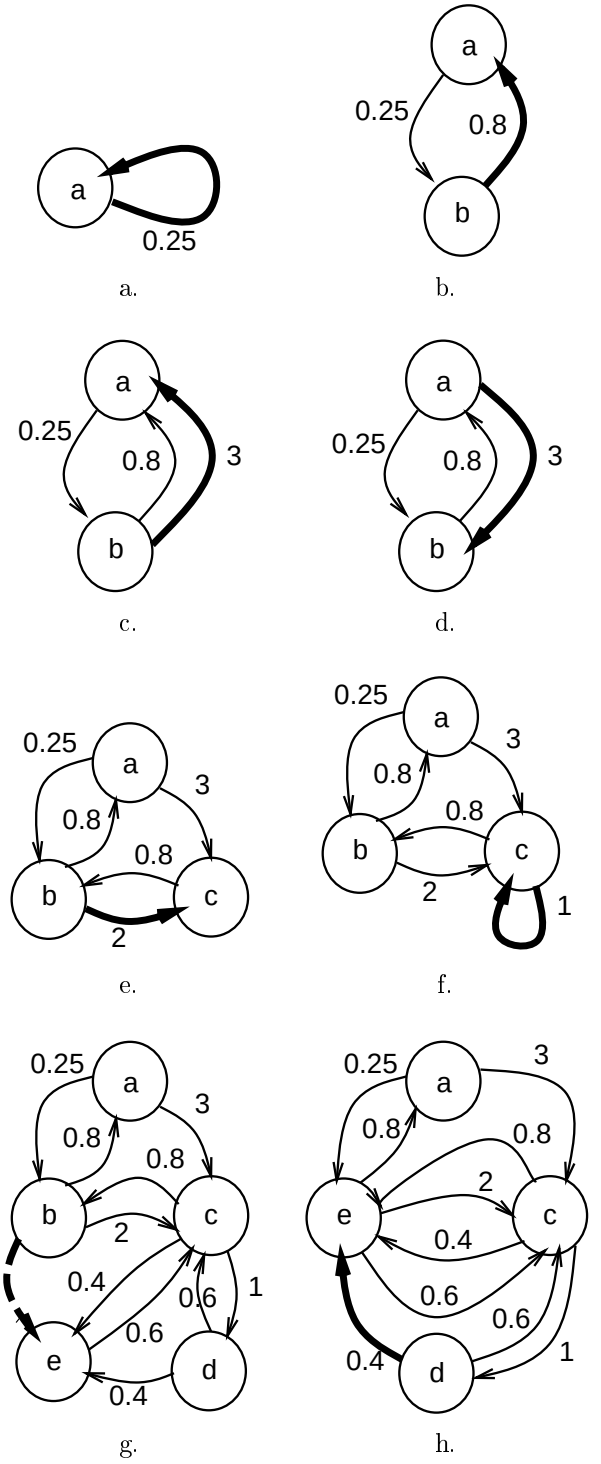


Figure 2: Constructing A Network

- set-function(n) – Changes the transfer function of the to-neuron in the current link, B, with: 0, for sigmoid; 1, linear; and 2, for oscillator.

- split(n) – Creates a new neuron, C, with a sigmoid transfer function, and moves the current link from A to C and creates a new link connecting from neuron C to neuron B with weight n.

The sequence of networks in figure 2 are intermediate networks in parsing, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) split(0.6) duplicate(0.4) split(0.6) duplicate(0.4) reverse parent(1) merge(1)*. Networks start with a single neuron, a, with an *oscillator* function, which has a single link, of weight 0.25, feeding to itself, figure 2.a. The sequence of intermediate networks after: *split(0.8)*, is b; *duplicate(3)*, is c; *reverse*, is d; *split(0.8) duplicate(2) reverse*, is e; *loop(1)*, is f; *split(0.6) duplicate(0.4) split(0.6) duplicate(0.4)* (which results in the current link being a virtual link from neurons b to e), is g; and after *merge(1)*, the final network is shown in h.

Neurons in the network are initialized to an output value of 0.0 and are updated sequentially by applying a transfer function to the weighted sum of their inputs with their outputs clipped to the range ± 1. The different transfer functions are: a sigmoid, using $tanh(sum\ of\ inputs)$; linear; and an oscillator. The oscillator maintains a state value, which it increases by 0.01 each update. Its output is its state value plus the weighted sum of its inputs mapped to a triangle-wave function with period 4 and a minimum of -1 and maximum of 1. Use of an oscillator increases the bias towards networks whose outputs cycle over the sigmoid-only networks used in [Komosinski & Rotaru-Varga, 2000, Lipson & Pollack, 2000] but is a more simple model than that of [Sims, 1994] which had a variety of transfer functions and oscillating neurons.

2.3 COMBINING BODY AND BRAIN

To simultaneously create a creature's neural controller and morphology, the languages for constructing a neural network and for constructing a body are combined. When processing the command string, a neural-construction command affects the construction of the neural controller and a morphology-construction command affects the construction of the body, with a few modifications. *Push* and *pop* operators, '[' and ']' are used to store and retrieve the current construction state, which now consists of the current link and the current location and orientation on the body. To give the neural controller control of the body, each time a joint command (*revolute-1, revolute-2, twist-90 or twist-180*) is executed, the neural-command *output(1)* is also called. This output neuron then controls the joint angle of the actuated joint.

Once a string of build commands has been executed and the resulting creature is constructed, its behavior is evaluated in a quasi-static kinematics simulator, similar to that used by [Lipson & Pollack, 2000]. First the neural network is updated to determine the desired angles of each actuated joint. Then the kinematics are simulated by computing successive frames of moving joints in small angular increments of at most 0.06°. After each update the structure is then settled by determining whether or not the creature's center of mass falls outside its footprint and then repeatedly rotating the entire structure about the edge of the footprint nearest the center of mass until it is stable.

2.4 PARAMETRIC L-SYSTEMS

The strings of build commands are generated by a context-free, parametric Lindenmayer-system (P0L-system). L-systems are a grammatical rewriting system introduced to model the biological development of multicellular organisms [Lindenmayer, 1968]. Rules are applied in parallel to all characters in the string just as cell divisions happen in parallel in multicellular organisms. For example, the L-system,

$$\begin{aligned} a: &\quad \rightarrow a\ b \\ b: &\quad \rightarrow b\ a \end{aligned}$$

if started with the symbol a, produces the following strings,

$$a$$
$$ab$$
$$abba$$
$$abbabaab$$

A parametric L-system [Lindenmayer, 1974] is a class of L-systems in which production rules have parameters and algebraic expressions can be applied when parameter values to successors. Parameter values can also be used in determining which production rule to apply. For example, the P0L-system,

$$\begin{aligned} a(n): &\quad (n > 1) &\rightarrow a(n-1)\ b(n) \\ a(n): &\quad (n \leq 1) &\rightarrow a(0) \\ b(n): &\quad (n > 2) &\rightarrow b(n/2)\ a(n-1) \\ b(n): &\quad (n \leq 2) &\rightarrow b(0) \end{aligned}$$

When started with $a(4)$, the P0L-system produces the following sequence of strings,

$$a(4)$$
$$a(3)b(4)$$
$$a(2)b(3)b(2)a(3)$$
$$a(1)b(2)b(1.5)a(2)b(0)a(2)b(3)$$
$$a(0)b(0)b(0)a(1)b(2)b(0)a(1)b(2)b(1.5)a(2)$$
$$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(1)b(2)$$
$$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)$$

Advantages of a parametric L-system are a P0L-system can produce a family of structures, with the specific structure created being determined by the starting parameters. Similarly, parameters can be used so that repeating patterns of connections will have different weights. An example of a P0L-system for a network is,

$P0(n0)$:
$n0 > 1.0 \rightarrow \quad P1(n0)\ P0(n0-1)$
$n0 > 0.0 \rightarrow \quad loop(1)\ P1(1)\ parent(1)\ merge(1)$

$P1(n0)$:
$n0 > 1.0 \rightarrow \quad split(0.8)\ duplicate(n0)\ reverse$
$n0 > 0.0 \rightarrow \quad \{split(0.6)\ duplicate(0.4)\}(2)\ reverse$

This L-system consists of two productions, each containing two condition-successor pairs and when started with P0(3) produces the sequence of four strings: *a*, *P1(3) P0(2)*; *b*, *split(0.8) duplicate(3) reverse P1(2) P0(1)*; *c*, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) P1(1) parent(1) merge(1)*; and *d*, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) {split(0.6) duplicate(0.4)} reverse parent(1) merge(1)*. This last is interpreted as: *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) split(0.6) duplicate(0.4) split(0.6) duplicate(0.4) reverse parent(1) merge(1)*, and the network that it constructs is shown in figure 2.h.

2.5 EVOLUTIONARY ALGORITHM

The evolutionary algorithm used to evolve L-systems is the same as [Hornby & Pollack, 2001]. The initial population of L-systems is created by making random production rules. 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 by applying mutation or recombination. Mutation creates a new individual by copying the parent individual and making a small change to it. 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; or changing the condition equation. 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 $p2$.

3 EXPERIMENTAL RESULTS

In this section we present results in evolving 3D locomoting creatures using both oscillating joints as controllers and using neural networks as controllers. To evolve creatures that locomote we set their fitness to be a function of the distance moved by the creature's center of mass less the distance ground points were dragged along the ground – this penalty encourages creatures to evolve stepping or rolling motions over sliding motions.

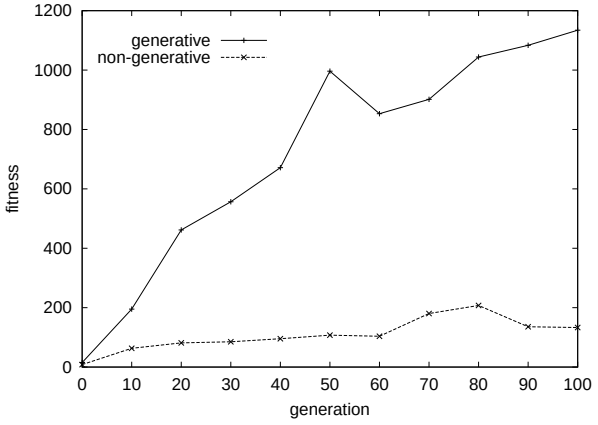
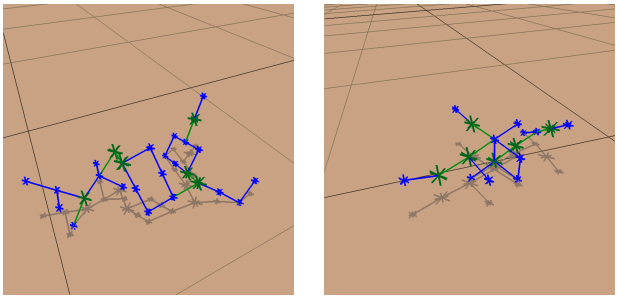


Figure 3: Generative vs. Non-generative Encodings

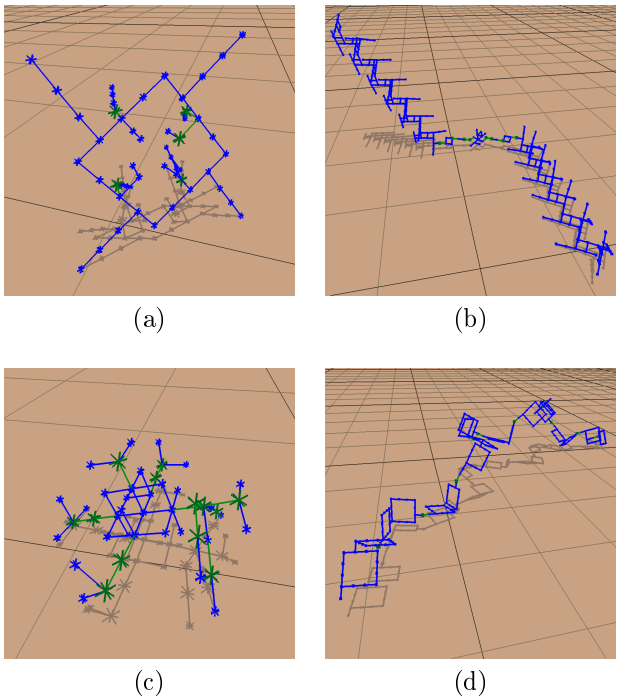


(a) (b)

Figure 4: Results with Non-generative Encoding

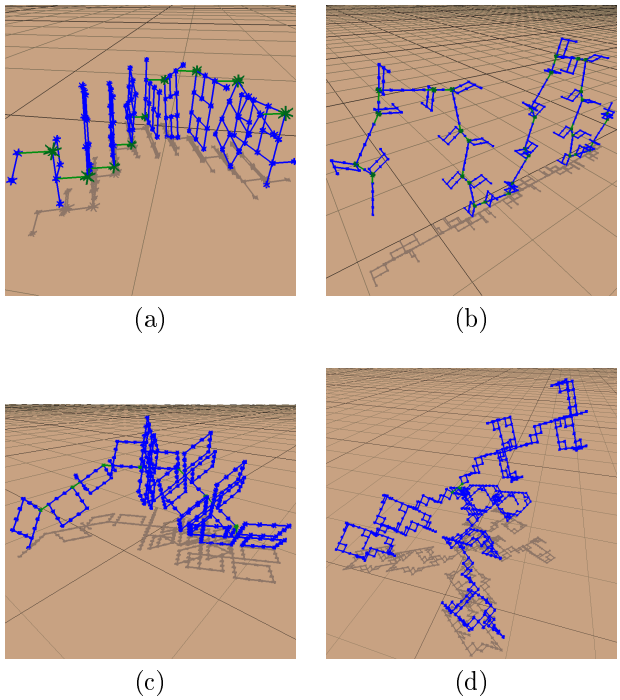
Initially we ran two sets of experiments to compare a generative encoding against a non-generative encoding. In these experiments we added the constraint that creatures could not have a sequence of more than 4 bars in a row that was not part of a cycle as a representative limit to physically plausible creatures while not providing any shaping bias[1]. The evolutionary algorithm was configured to run with a population of

[1]In a true dynamics simulator actual torques on joints would be calculated and then a constraint on the allowable torque could be used.



(a) (b) (c) (d)

Figure 5: Results with Generative Encoding



(a) (b) (c) (d)

Figure 6: Other Oscillator Creatures

100 individuals for 100 generations and morphologies had an upper limit of 350 bars. The non-generative encoding was allowed to use up to 10000 build commands and the generative encoding used 2 parameters and 15 production rules, with 2 condition-successor pairs for each production rule, with each successor having a maximum of 20 build commands. 10 trials were run with each encoding type, and the average of the fittest individual found at each generation is plotted in the graph in figure 3. Two individuals evolved using the non-generative encoding are shown in figure 4 and four individuals evolved using the L-system as a generative encoding are shown in figure 5. In addition to producing faster creatures, the L-system encoding produced creatures with greater self-similarity and had more parts – the average number of bars in the fittest creatures was 16 using the non-generative encoding and 120 with the L-system encoding.

Other evolutionary runs using the L-system encoding were made with different fitness functions and a higher upper limit on the number of allowed bars. The individuals in figure 6 were evolved against a fitness function that rewarded for having closed loops in the morphology – a is a sequence of rolling rectangles with 169 bars; b is an undulating serpent with 339 bars; c is an asymmetric rolling creature with 306 bars; and d is a four-legged walking creature with 629 bars.

Next we ran experiments combining the neural-network construction language from section 2.2 with morphology construction language of section 2.1 to evolve creatures with neural controllers. To encourage networks with complex dynamics to evolve, individuals were rewarded for the average number of inputs to hidden units and for the range in values of the output units. The evolutionary algorithm was configured to run with a population of 100 individuals for a maximum of 500 generations. Experiments were run using 20 production rules, 3 condition-successor pairs and 2 parameters for each production rule for which approximately half the runs produced interesting creatures Examples of evolved creatures are shown in figure 7: a has 49 neurons and moves by first stretching out its arms, then twisting its body as it closes up to move sideways; b has 41 neurons and moves by falling over each time it wraps up into a circle and unwraps; c has 24 neurons and moves by using the two lower squares as an arm to push it forward; d has 150 neurons and moves by coiling up into a circle to roll; and e has 19 neurons and moves by using its tail to roll it along like a wheel. The network of the creature in figure 7.f is shown in figure 8. In addition to being fairly regular, its linear sequence of outputs also corresponds to the linear sequence of joints in its morphology. It moves by twisting itself to roll sideways.

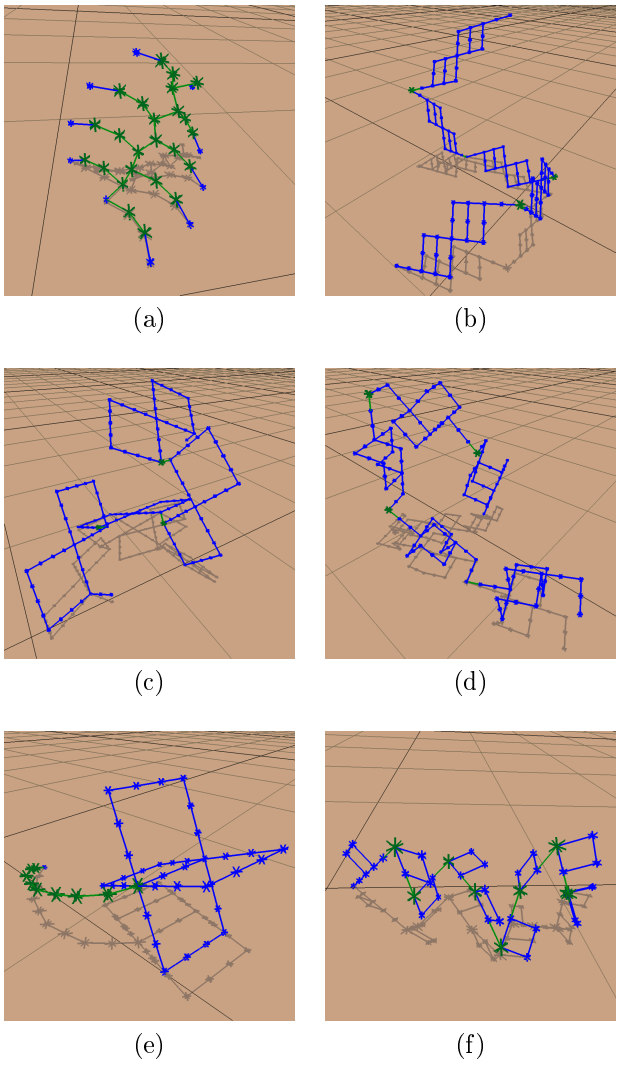


Figure 7: Neural Network Controlled Creatures

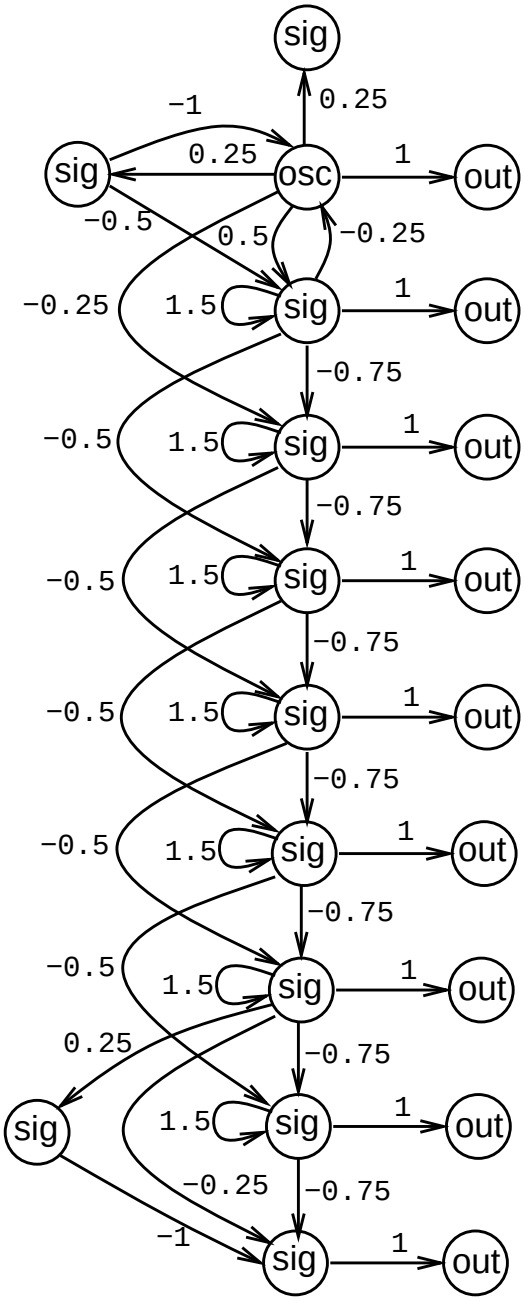


Figure 8: Evolved Neural Network

4 DISCUSSION

As the number of parts in a creature increase, so does the difficulty in hand-designing a controller for it. This system for evolving P0L-systems automatically produced creatures, and their controllers, with hundreds of parts and varying degrees of regularity.

We also used the results of one evolutionary run as the starting population for another run, the creature in figure 6.b is the result of one run seeded with a previously evolved creature. This method of using multiple runs of evolution is one way to use EAs to optimize and explore the design space of the input creature or to create creatures with similar locomotion styles. Alternatively, a simple creature can be designed by hand and the evolutionary system can improve upon it.

For this work we used parametric L-systems as a way to increase complexity over basic L-systems. Other types of L-systems that could have been used are probabilistic L-systems and context sensitive L-systems [Prusinkiewicz & Lindenmayer, 1990]. Probabilistic production rules do not have a condition part, rather they have multiple successors, each with a probability that it will be used to replace the predecessor. While good for generating a variety of similar struc-

tures, this system is non-deterministic – which makes it unsuitable for developing structures that need to be re-created the same each time. Another way in which variation can be applied to an L-system is through the addition of context. Context sensitive L-systems examine the characters to the left and right of the character to be rewritten to determine which successor to replace it with. While this class of L-systems is deterministic, not having parameters results in the inability to take advantage of parametric terminals, such as the oscillators whose parameter specifies the speed of oscillation. Using parameters also has the advantage of allowing one production rule to be used to generate a class of objects. In this way parameters are analogous to the arguments of a function in a computer program and the evolution of an L-system becomes like the evolution of a computer program, as in genetic programming [Koza, 1992].

5 CONCLUSION

An integrated encoding for generatively creating both creature morphology and neural controller was achieved by using evolutionary techniques to evolve P0L-systems. Using this system, the morphologies and controllers were evolved for locomoting creatures. Creatures evolved using the generative encoding moved faster than creatures evolved using the non-generative encoding. In comparison to related work, these evolved creatures consisted of an order of magnitude more parts and had a higher degree of regularity than [Sims, 1994, Komosinski & Rotaru-Varga, 2000, Lipson & Pollack, 2000].

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant, 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.

References

[Abelson & deSessa, 1982] Abelson, H. & deSessa, A. A. (1982). *Turtle Geometry.* M.I.T. Press.

[Boers & Kuiper, 1992] Boers, Egbert J. W. & Kuiper, Herman (1992). Biological metaphors and the design of modular artificial neural networks. Master's thesis, Leidea University, the Netherlands.

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

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

[Hornby & Pollack, 2001] Hornby, Gregory S. & Pollack, Jordan B. (2001). The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation.*

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

[Komosinski & Rotaru-Varga, 2000] Komosinski, M. & Rotaru-Varga, A. (2000). From directed to open-ended evolution in a complex simulation model. In Bedau, McCaskill, Packard, & Rasmussen (Eds.), *Artificial Life 7*, pp. 293–299.

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

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

[Lindenmayer, 1974] Lindenmayer, A. (1974). Adding continuous components to L-Systems. In Rozenberg, G. & Salomaa, A. (Eds.), *L Systems*, Lecture Notes in Computer Science 15, pp. 53–68. Springer-Verlag.

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

[Luke & Spector, 1996] Luke, Sean & Spector, Lee (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J. (Ed.), *Late-breaking Papers of Genetic Programming 96*, pp. 117–124. Stanford Bookstore.

[Prusinkiewicz & Lindenmayer, 1990] Prusinkiewicz, P. & Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants.* Springer-Verlag.

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