Creating High-Level Components with a Generative Representation for Body-Brain Evolution

Gregory S. Hornby*
Jordan B. Pollack
DEMO Lab
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
hornby@cs.brandeis.edu
pollack@cs.brandeis.edu

Abstract One of the main limitations of scalability in body-brain evolution systems is the representation chosen for encoding creatures. This paper defines a class of representations called *generative representations*, which are identified by their ability to reuse elements of the genotype in the translation to the phenotype. This paper presents an example of a generative representation for the concurrent evolution of the morphology and neural controller of simulated robots, and also introduces *GENRE*, an evolutionary system for evolving designs using this representation. Applying GENRE to the task of evolving robots for locomotion and comparing it against a non-generative (direct) representation shows that the generative representation system rapidly produces robots with significantly greater fitness. Analyzing these results shows that the generative representation system achieves better performance by capturing useful bias from the design space and by allowing viable large scale mutations in the phenotype. Generative representations thereby enable the encapsulation, coordination, and reuse of assemblies of parts.

Keywords
Body-brain evolution, generative representations, representation, Lindenmayer systems (L systems)

1 Introduction

The evolution of artificial creatures has come a long way since Dawkins' evolution of two-dimensional shapes [8]. Controllers have been evolved for fixed morphologies: first with stimulus-response rules for animated, articulated creatures [28, 35]; then with neural controllers [10]; and more recently for the dynamic gait of a physical, quadruped robot [11]. More true to the spirit of artificial life is the evolution of both body and brain, starting with Sims' evolution of block creatures—for swimming, walking, and light seeking [33], as well as competing for the possession of a box [32]—and Ventrella's evolution of stick figures for walking [36]. This has been followed by the evolution of walking creatures [20, 26, 6], summarized in [34]. For the most part, this newer work has not managed to surpass the complexity of Sims' original block creatures. We propose that one source of this difference is the differences in representations used to encode creatures. This paper identifies a class of representations, called *generative representations*, and investigates their impact on the problem of evolving locomoting robots.

Here we consider genotypic representations as a kind of programming language. With this analogy, the fundamental properties of programming languages can be used to

* Corresponding author.

understand and classify different approaches to the underlying representations of evolutionary systems. From [2], the fundamental elements of programming languages are:

- *Combination.* Languages create the framework for the hierarchical construction of more powerful expressions from simpler ones, down to atomic primitives.
- *Control flow.* All programming languages have some form of control of execution, which permits the conditional and repetitive use of structures.
- *Abstraction.* Both the ability to label compound elements (to manipulate them as units) and the ability to pass parameters to procedures are forms of abstraction.

In implementation, these elements can be parceled out to different mechanisms, such as branching, variables, bindings, and recursive calls, but they are nonetheless present in some form in all programmable systems. Some of these basic properties have also been shown to have analogues in biological systems: phenotypes are specified by combinations of genes; the expression of one gene can be turned on or off by the expression of another gene [24]; and an upstream protein can control a downstream protein's activity through a signaling pathway [3].

We can use the properties of programming languages to understand and classify design representations. A fundamental distinction is whether a representation used in evolution is *direct* or *generative*. In a direct representation there is a one-to-one mapping from each representational element of the genotype to some component in the phenotype. A generative representation, on the other hand, is one which is capable of reusing elements of the genotype in the construction of phenotypic components. This reuse of components can come from iteration or from abstraction. Continuing with the programming language analogy, a generative representation is a kind of language such that heritable genotypic elements, together with a translation or compilation process, control the expression of phenotypic components. For instance, the processes may interpret genotypic information as constructs (loops, procedure calls, variables, parameters, etc.) to control genetic expression. Thus with a generative representation, the individuals in the population are programs in a language whose instructions control the flow of construction commands for creating each design.

Here we use a Lindenmayer system (L-system) [25] as the generative representation to encode creatures. L-systems are a grammatical rewriting system in which the rewriting rules are applied in parallel to all symbols in a string. Figure 1a shows a graphical representation of the rules for our generative representation. In these images cubes represent procedure calls, black spheres represent conditionals, triangles represent the repeat operator, and spheres represent construction commands. Figure 1b shows the sequence of assembly strings generated by this set of rules. The sequence begins with the first cube (here a blue and red one), and the sequence of strings below it are the strings generated after each iteration of parallel replacement. The last string of symbols is the assembly procedure used for constructing a robot.

In this paper we explain the details of this generative representation system and show that during evolution the system can incorporate useful biases from a robot design problem. This incorporation of bias comes through reuse of elements in the genotype. By continually creating new components and including these in the variations available, the unit of variation scales with the complexity of the design. Because changing the definition of a reused assembly of parts results in a change in all occurrences of that assembly, the design system has the ability to make coordinated changes in several parts of a design simultaneously.

The system we use for the evolution of robots is called *GENRE*, which stands for "generative representations." This system has been used to evolve tables [14] and

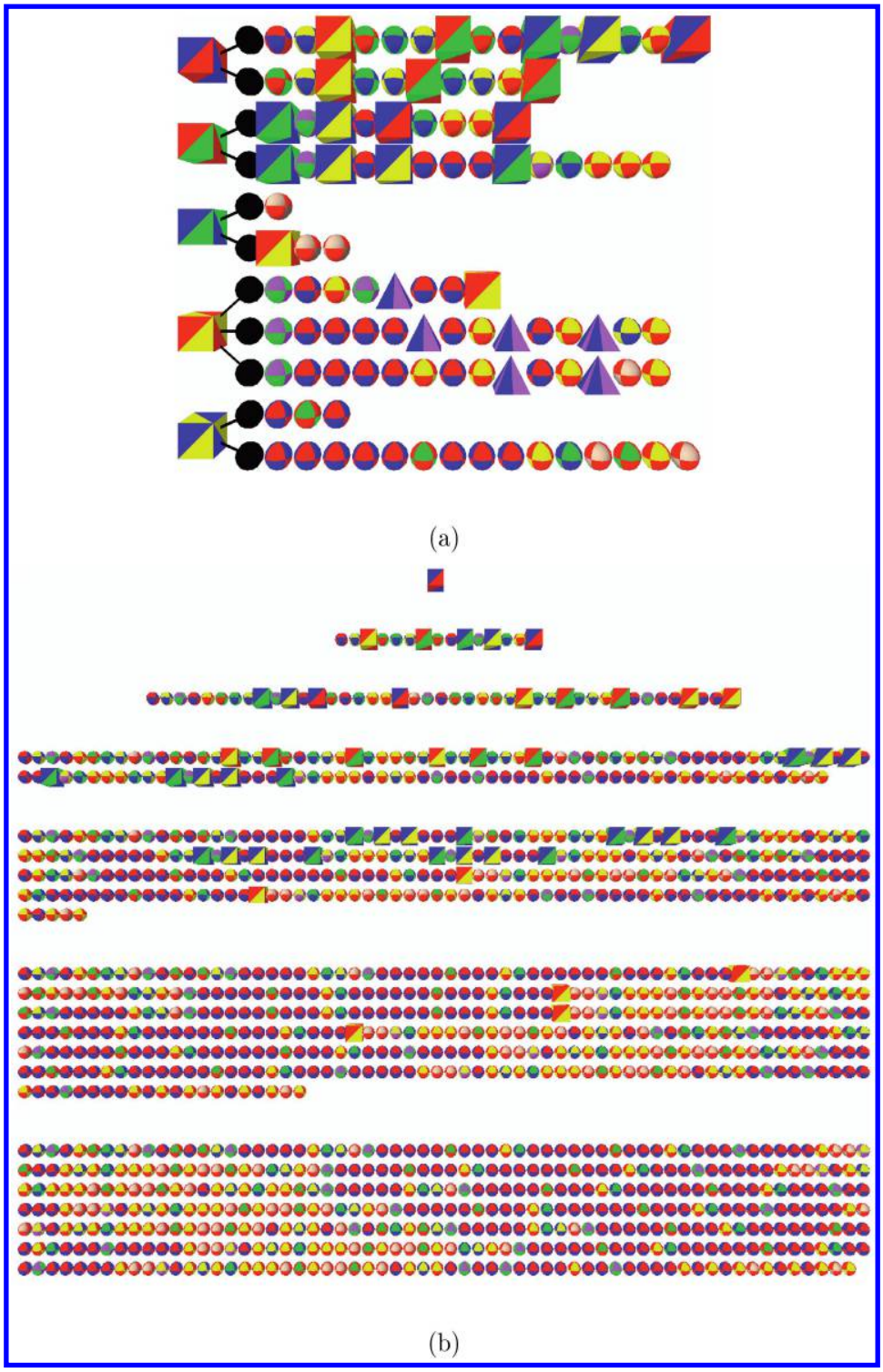


Figure 1. (a) A graphical version of the generative representation, along with (b) the sequence of assembly strings it produces.

oscillator-controlled robots [12, 15], demonstrating in both cases that an evolutionary algorithm (EA) using a generative representation outperforms an EA using a direct representation. Here we describe the application of GENRE to neural-network-controlled robots, and we call our robots *genobots* (for generatively encoded robots). Moving to neural networks allows us to generate more complex movement patterns and allows for later work to include sensors in order to evolve robots with reactive controllers. With GENRE we evolve neural-network-controlled locomoting robots and compare their search performance against a system that uses a direct representation. We find that the generative representation system rapidly produces robots with significantly greater fitness. Analyzing these results shows that the generative representation system achieves better performance by capturing useful bias from the design space and by allowing viable large scale mutations in the phenotype. Generative representations thereby enable the encapsulation, coordination, and reuse of assemblies of parts.

The rest of the paper is organized as follows. First we review the different representation systems used by body-brain evolution systems. Then we describe the three parts of our body-brain evolution system: the compiler for the generative representation, the body-brain constructor and simulator, and the evolutionary algorithm. Next we present the results of our experiments in evolving locomoting creatures. This is followed by a discussion of our findings and conclusions.

2 Review of Other Representations

Prior methods of body-brain evolution each have their own format for representing the creatures being evolved. In this section we first describe how the properties of programming languages apply to design representations and then review several representative systems.

With the exception of combination, the elements of programming languages listed in the introduction translate directly to design representations. Two types of control flow are conditionals and iterative expressions. Conditionals can be implemented with an `if` statement, as in genetic programming (GP) [22], or a rule that governs the next state in a cellular automaton (CA). Iteration is a looping ability, such as the repeat structure in cellular encoding [9], or embedded in the fundamental behavior of CAs. Abstraction is the ability to encapsulate part of the genotype and label it so that it can be used like automatically defined functions (ADFs) in GP [23] or automatically defined subnetworks in cellular encoding. Abstraction can be seen when subfunctions can take parameters, as with ADFs. Combination refers to the ability to create more complex expressions from the basic set of commands in the language. While GP allows explicit combinations of expressions, combination is not fully enabled by mere adjacency or proximity in the strings utilized by typical representations in genetic algorithms.

Sims used an embedded, directed graph representation to specify the construction of his creatures [33]. Nodes in the top layer of the graph represent body segments, and within the node is another graph for the body segment's neural controller. An advantage of encapsulating the neural units inside the nodes for body segments is that copying, or recombining, subgraphs automatically swaps the associated neural controller for a section of body parts. This representation is generative because cycles in the graph, along with a recursive-limit parameter, are procedural constructs that specify the number of times nodes in the cycle are to be traversed in the construction phase. But the two-layer structure does not allow a repetition of the neural processing units inside a body segment, because they are directly encoded as a design inside a body node.

The stick creatures evolved in Ventrella's work [36, 37] are encoded as fixed-length vectors of parameters for constructing a creature. Parameters specify the number of segments for a central backbone, the number of opposing limbs, the number of segments

Table 1. Properties of the different representations.

	Control flow		Abstraction	
System	Iteration	Conditionals	Labels	Parameters
Bongard and Pfeifer [6]	Yes	Yes	No	No
Framsticks—*recur*/*simul* [20]	No	No	No	No
Framsticks—*devel* [21]	Yes	No	No	No
GENRE—direct	No	No	No	No
GENRE—generative	Yes	Yes	Yes	Yes
GOLEM [26]	No	No	No	No
Sims [33]	Yes	No	No	No
Ventrella [36]	Yes	No	No	No

in each limb, joint angles, and details for the oscillator network. While this representation is generative in the sense that it allows reuse of the genotype, the structure of what can be reused is fixed and not evolvable.

The genotypes of creatures in Framsticks [20] are encoded as a linear assembly procedure for constructing a creature, with bracketing, which turns the basic structure from a string to a tree. Commands in the command set attach sticks to existing ones as well as construct the neural controller—a command for creating a neuron attaches it to the stick most recently created and is then followed by a sequence of link connections. More recently Komosinski and Rotaru-Varga have compared their original representation, called *recur* for direct recurrent, against the actual representation used by the simulator, *simul*, and a tree-structured representation, called *devel* for developmental [21]. Simul consists of a list of all objects (sticks, joints, neurons, sensors, and actuators) that make up a creature, along with all of that object's attributes. *Devel* is a tree-structured version of *recur* with iteration through a repeat node for repeating a subtree. Of these, only *devel* is generative, because it is the only representation that allows for reuse of the genotype.

In GOLEM [26], the representation of a creature is the design itself. Both the morphology and the neural controller are stored as graph-based data structures with links connecting actuated joints to neurons in the network. One challenge in using a graph-based representation is in implementing meaningful recombination operators between graphs. In this case, mutation was the only variation operator implemented.

The genotypes in the work of Bongard and Pfeifer [6] are a set of gene expression rules for growing creatures under a simulated ontogenetic process. These rules determine the division of body segments based on simulating chemical concentrations inside each segment. Each segment also contains a neural controller, which is developed by the gene expression rules using cellular encoding commands [9]. Bongard and Pfeifer report that similar parts of a creature also have similar gene expression patterns, suggesting that this method can produce modular creatures. Here reuse comes about through an iterative loop external to the evolved representation; at each update iteration gene rules are applied to the developing creature.

The properties of the different representations used for the evolution of a robot's morphology and controller are summarized in Table 1. Not included in this summary is a column for a representation's structure, which is left out because determining the structure of a representation is somewhat subjective and also depends on how the variation operators act on an encoding. Of the reviewed representations, only Genre's generative representation (described in the next section) has reuse through

both iteration and parameterized procedures. Whereas iteration produces exact copies of the repeated genotype, parameterized procedures can act as a parameterized module, with the resulting phenotype depending on the input parameters.

3 Evolutionary System

The evolutionary system used to evolve creatures consists of the robot constructor and simulator, the compiler for the generative representation, and the evolutionary algorithm. Each robot is constructed from a sequence of construction commands, called an assembly procedure, which specifies how to assemble both the morphology and the robot's neural controller. This string of construction commands is either evolved directly or produced by compiling the generative representation. Robots encoded with a generative representation are called *genobots* (for generatively encoded robots). Our system uses Lindenmayer systems (L-systems) as the generative representation for the genobots. The evolutionary algorithm evolves a population of these L-systems, using the fitness returned by the robot simulator. The following subsections describe each of these parts.

3.1 L Systems as a Generative Representation Language

The generative representation for each genobot is an L-system, a grammatical rewriting system introduced to model the biological development of multicellular organisms [25]. Rules are applied in parallel to all characters in the string, just as cell divisions happen in parallel in multicellular organisms. A basic L-system consists of a collection of rewriting rules, such as

$a: \rightarrow b,$

$b: \rightarrow a\ b.$

When started with the symbol a, this L-system produces the following sequence of strings:

a

b

ab

bab

$abbab$

$bababbab$

The class of L-systems used as the genotype for creatures in this work is context-free, parametric Lindenmayer systems (PL-systems). Context-free indicates that the rules for rewriting symbols do not depend on the symbol's neighbors, and parametric specifies that symbols and rewriting rules can take parameters. Production rules consist of a rule head, which is the symbol to be replaced, followed by a number of condition-successor pairs. The condition is a Boolean expression in the parameters of the production rule, and the successor (also called the production body) consists of a sequence of characters that replace the rule head. Rule-head symbols are rewritten by testing each of their conditions sequentially and replacing the rule-head symbol with the successor of the first condition that succeeds.

Because the PL-system does not have the ability to directly repeat a block of symbols, iteration is added through a block-replication command. Symbols of the form

{ *block* }(*n*) are repeated *n* times, and are similar to *for-next* loops in computer programs. The resulting representation language has the properties of iteration, conditionals, and abstraction with parameters.

Previously EAs have been combined with L-systems to evolve neural networks [19, 5], plants [17, 29] and architectural structures [7]. For the most part, this past work has used non-parametric L-systems, whereas here we use parametric ones. An advantage of a parametric L-system over a non-parametric one is that a given PL-system can produce a family of strings, with the specific string determined by the starting parameter(s). For example, a parameter of a production rule can be used as the argument of the repeat command to specify the number of times a substring is to be repeated. Furthermore, parametric L-systems naturally allow for parametric commands in the language—the parameter of a network construction command can specify the weight of a newly created link in the network.

3.2 Robot Constructor

Robots are constructed through a method that is a synthesis of Logo-style turtle graphics [1] and cellular encoding [9]. Commands in the assembly procedure are for constructing the robot's morphology and its neural controller. So that body and brain are joined, the command for the creation of an actuated joint also creates a link to a neuron in the neural network. The following sections describe the morphology and neural network constructors separately and then how a robot's body and brain are simultaneously constructed.

3.2.1 Network Constructor

The method for constructing the neural controllers for the artificial creatures is based on that of cellular encoding [9]. The main difference is that build commands operate on the links connecting the nodes, as with edge encoding [27], instead of on the nodes of the network. With edge encoding at most one link is created with a network construction command, which allows each command to also specify the weight to attach to that link, and subsequences of build commands will construct the same subnetwork, regardless of their location in the assembly procedure. Another distinction between this and cellular encoding is that assembly procedures for constructing networks are linear sequences of commands (strings) and not trees. A branching ability is added to strings by using bracketed L-systems [25] with *push* and *pop* operators for storing and retrieving the current link to a stack.

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 state—consisting of the from neuron, the to neuron, and the indices of the links into these neurons—to and from the stack. This stack of edges allows a form of branching to occur in the representation: 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. The commands for this language are listed in Table 2, for which the current link connects from neuron A to neuron B.

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: sigmoid, using tanh(sum of inputs); linear; and oscillator. Oscillator units maintain a state that is increased by 0.01 after each update. The output of an oscillator unit is mapped to the range -1 to 1 by applying a triangle wave function, with a period of four, to the sum of its inputs and its state. While using oscillating neurons increases the bias for simple networks with simple oscillating patterns over the sigmoid-only networks used

Table 2. Command set for constructing neural networks.

Command	Description
( )	Pushes and pops link state to and from stack.
decrease-weight(n)	Subtracts n from the weight of the current link. If the current link is a virtual link, creates it with weight $-n$.
duplicate(n)	Creates a new link from neuron A to neuron B with weight n.
increase-weight(n)	Adds n to the weight of the current link. If the current link is a virtual link, 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 A as an input with B. The current link then becomes the nth input into 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.
set-function(n)	Changes the transfer function of the to-neuron in the current link, B, to sigmoid ($n = 0$), linear ($n = 1$), or oscillator ($n = 2$).
split(n)	Creates a new neuron, C, with a sigmoid transfer function, moves the current link from A to C, and creates a new link connecting C to B with weight n.

in [20, 26], it yields a less biased model than that of [36], in which all actuators are driven by oscillators, or [33], which used a variety of transfer functions and oscillating neurons.

An example of the construction of a network using this system is shown in Figure 2, which contains the intermediate networks in parsing the following assembly procedure:

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*, which has an oscillator transfer function, and a single link of weight 0.25 feeding to itself (Figure 2a). After executing *split(0.8)*, a second neuron is created with a link of 0.8 to the oscillating neuron and the original link of weight 0.25 feeding into it (Figure 2b). Executing *duplicate(3)* creates a second

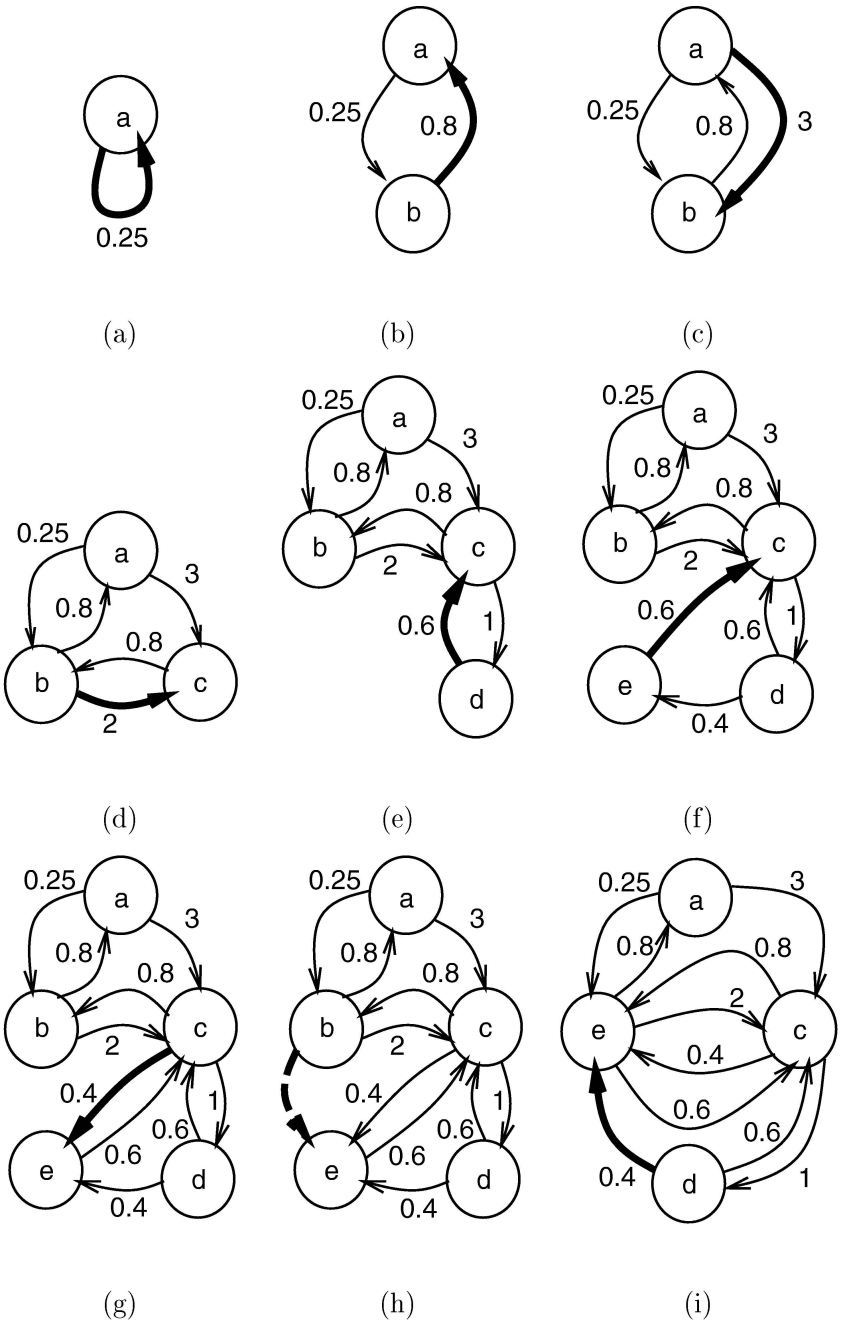


Figure 2. Construction of a neural network.

link from the second neuron to the first, which is then reversed in executing *reverse* (Figure 2c). The execution of *split(0.8) duplicate(2) reverse* creates a third neuron (Figure 2d). A link from the third neuron to itself with weight 1 is created by *loop(1)*, with another neuron created by *split(0.6)* (Figure 2e). This is followed by *duplicate(0.4)*, which creates an additional link from neuron c to d, and then neuron e is created with *split(0.6)* (Figure 2f). Another link is created from e to c with *duplicate(0.4)*, which is then reversed (*reverse*, Figure 2g). *Parent(1)* causes a shift of link state from the $c \rightarrow e$ link to a new *virtual link* $b \rightarrow e$, shown as a dashed line (Figure 2h). These two neurons are then joined together by the *merge(1)* command, and the final network is shown in Figure 2i.

3.2.2 Morphology Constructor

The morphology constructor uses a set of construction commands similar to that of L-system languages for creating plants [30] to build a body through the control of a Logo-style turtle [1]. As the turtle moves, rods are created, and these become the body of the robot. Commands instruct the turtle to move forward or backward, change orientation, or create an actuated joint. The method for constructing the morphology of a creature, with joints controlled by actuators, is described in [15]. The following section describes the method by which the morphology construction is combined with the neural network constructor of the previous section.

3.2.3 Neural-Network-Controlled Robots

A robot's morphology and neural controller are constructed by combining the command sets for constructing body and brain into one language and then building body and brain simultaneously. This command language consists of the morphology construction commands, listed in Table 3, and the neural construction commands from Section 3.2.1. The resulting language has two push-pop commands with two stacks: ( ), for pushing and popping the link state to the link stack; and [ ], for pushing and popping both the morphology and link states to a stack. A robot's body and brain are joined together by attaching the current input neuron to the newly created actuated joint each time a joint command—*revolute-1*, *revolute-2*, *twist-90*, or *twist-180*—is executed. By defining joint-creation commands in a way that affects both controller and morphology, we induce a connection between body and brain.

An example of an assembly procedure using this language is

[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward]
duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4)
reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0)
forward right(1.0) forward

A sequence of images showing intermediate stages in the construction of this robot is contained in Figure 3. Before any commands are processed, a robot consists of a single oscillating neuron and a point (Figure 3a). After executing the commands *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward]*, the robot consists of a square of four rods and the oscillating neuron (Figure 3b). After executing *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, a second neuron is created, and it is attached to the actuated joint at the end of the newly created rod (Figure 3c). The commands *duplicate(0.25) split(0.4) reverse revolute-1(1.0)* are repeated, and a third neuron is created and is attached to another actuated joint (Figure 3d). The last commands, *left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0)*

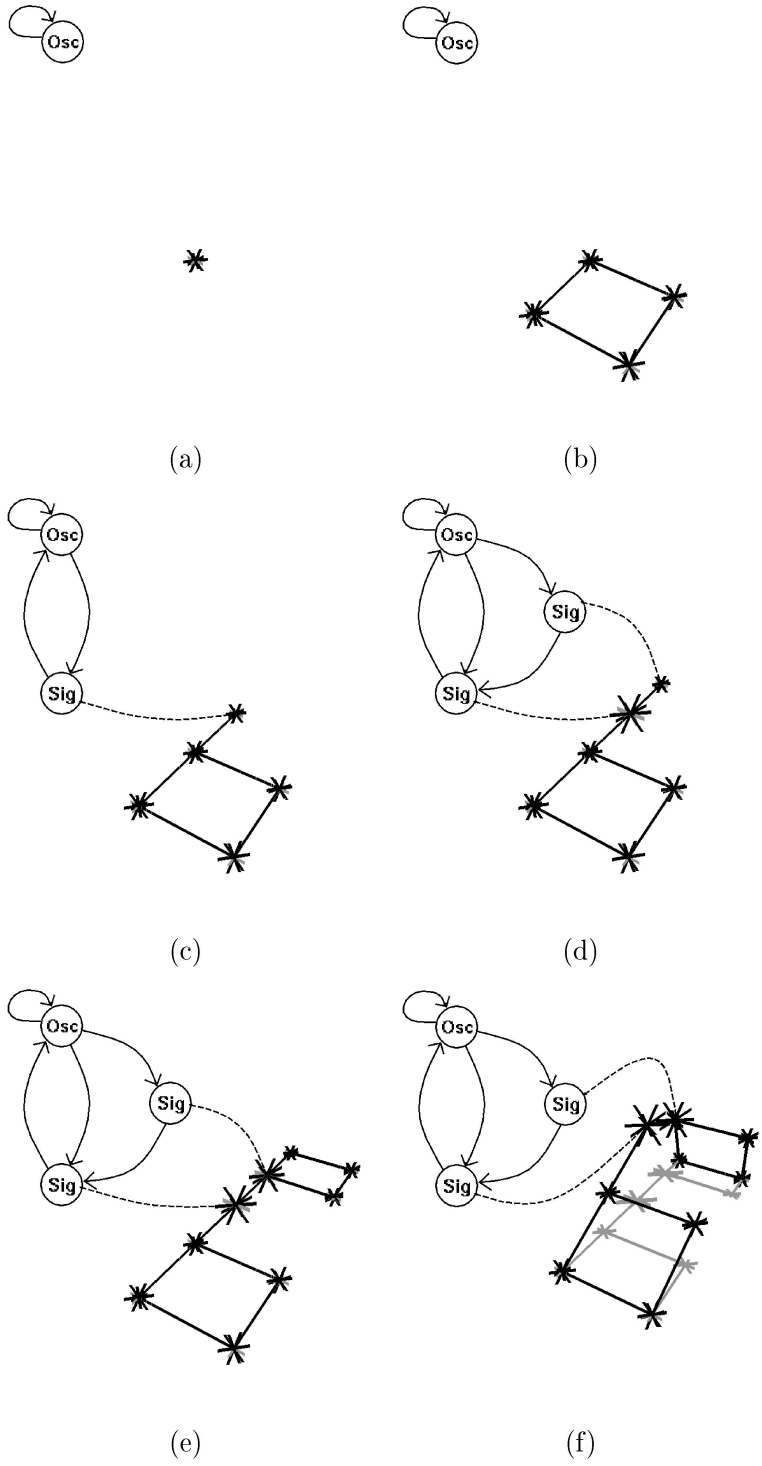


Figure 3. Construction of a genobot.

Table 3. Command set for constructing the morphology of a robot. Neural controllers are constructed by using this language along with that of Table 2.

Command	Description
[ ]	Pushes and pops state to and from stack.
forward	Moves the turtle forward in the current direction, creating a rod 10 units long if none exists or traversing to the end of the existing bar.
back	Goes back up the parent of the current bar.
revolute-1	Forward, ends with a joint with range 0° to 90° about the current Z axis that is controlled by the current neuron.
revolute-2	Forward, ends with a joint with range −45° to 45° about the current Z axis that is controlled by the current neuron.
twist-90	Forward, ends with a joint with range 0° to 90° about the current X axis that is controlled by the current neuron.
twist-180	Forward, ends with a joint with range −90° to 90° about the current X axis that is controlled by the current neuron.
left(n)	Rotates heading $n \times 90°$ about the turtle's Y axis.
right(n)	Rotates heading $n \times -90°$ about the turtle's Y axis.
up(n)	Rotates heading $n \times 90°$ about the turtle's Z axis.
down(n)	Rotates heading $n \times -90°$ about the turtle's Z axis.
clockwise(n)	Rotates heading $n \times 90°$ about the turtle's X axis.
counter-clockwise(n)	Rotates heading $n \times -90°$ about the turtle's X axis.

forward, attach another square onto the end of the last revolute-1 joint (Figure 3e). Figure 3f shows the creature with the joints halfway through their movement range.

An example of a generative representation using this construction language is

$$P0(n0): \quad n0 > 3.0 \rightarrow P1(5.0)\ P0(n0 - 2.0)\ \mathit{left}(1.0)\ P1(4.0)$$
$$n0 > 0.0 \rightarrow \{\ \mathit{duplicate}(0.25)\ \mathit{split}(0.4)\ \mathit{reverse}\ \mathit{revolute}(1.0)\ \}(2.0)$$

$$P1(n0): \quad n0 > 4.0 \rightarrow [\ P1(4.0)\ ]$$
$$n0 > 0.0 \rightarrow \{\ \mathit{right}(1.0)\ \mathit{forward}(1.0)\ \}(n0)$$

This L-system consists of two productions, each containing two condition-successor pairs, and, when started with P0(4), produces the following sequence of strings[1]:

1. *P0(4)*
2. *P1(5.0) P0(2.0) left(1.0) P1(4.0)*
3. *[P1(4.0)]{ duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0){ right(1.0) forward(1.0) }(4.0)*
4. *[{ right(1.0) forward(1.0) }(4.0)]{ duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0){ right(1.0) forward(1.0) }(4.0)*

1 For clarity the unraveling of block replication expressions is left until the final iteration.

5. *[right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward] duplicate(0.25) split(0.4) reverse revolute-1(1.0) duplicate(0.25) split(0.4) reverse revolute-1(1.0) left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*

This last sequence of commands is the same assembly procedure as the one that produces the genobot in Figure 3.

3.2.4 Simulation

Once a string of build commands has been executed and the resulting robot is constructed, its behavior is evaluated in a quasi-static kinematics simulator similar to that used in [26]. 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 robot'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.

To achieve robot designs that are sufficiently robust to transfer to the real world, error is added to evolved structures as in the method of [18] and [16]. A robot design is evaluated by simulating it three times, once without error and twice with different error values applied to the joint angles. Error is applied to all connections that are not part of a cycle and is a random rotation in the range of ± 0.1 radians about each of the three coordinate axes. The returned fitness of an evolved individual is the minimum fitness scored from the three trials. By implementing error that is fixed throughout a trial and evaluating a design with different error values, evolved designs are made robust to imperfections in real-world construction. Examples of oscillator-controlled robots that were successfully transferred to the real world are in [12, 13].

3.3 Evolutionary Algorithm

The evolutionary algorithm and variation operators are described in detail in [15]; here we give an overview of the system. 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 as 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 in it. Changes that can occur are: replacing one command with another; perturbing the parameter of a command by adding or subtracting a small amount to or from it; changing a production rule's parameter equation in a successor; adding or 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 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 subsequence of commands from a successor in $p2$ into c, or replacing a subsequence of commands in a successor of c with a subsequence of commands from a successor in $p2$. Data is kept for each individual L-system—specifically, which production rules and successors were used, as well as the value range for each parameter. This data, similar to the environment frame of a programming language, allows variation operators to be applied only to those production rules that were used. It also allows history-based constraints on the mutation of conditional values.

Since variations sometimes create an invalid robot (with too many or too few rods, or such that the body parts intersect at some point while moving), variation operators

are tried a second time, for a particular set of parents, if the first attempt did not create an offspring whose fitness was at least 10% of that of its parent(s).

4 Results

To compare a direct representation with a generative representation we evolved neural-network-controlled robots for the task of locomotion. The fitness was a function of the distance moved by the robot's center of mass on a flat surface. In order to discourage sliding, the fitness was reduced by the distance that points of the robot's body were dragged along the ground. Finally, a design was given zero fitness if it had a sequence of four or more rods in which none of the rods was part of a closed loop with other rods. This constraint was intended to keep the system from producing spindly robots that would not function well in reality.[2] The evolutionary algorithm was configured to run with a population of 200 individuals for 250 generations. The direct representation was implemented as an L-system with one production rule, no arguments, and one condition-successor pair whose condition always succeeds, and without the repeat operator or the ability to call production rules. The maximum length of the production body was set to 10,000 commands, allowing assembly procedures of up to 10,000 commands to be evolved. The generative representation used an L-system with fifteen production rules, two condition-successor pairs, and two parameters for each production rule. For the generative representation, the maximum length of the production body was set to fifteen commands, and the maximum allowed length of an unraveled generative representation was set to 10,000 commands—the same length as with the direct representation. Implementing the direct representation as a degenerate case of the generative representation allowed the evolutionary design system to use the same variation operators on both representations, so that the only difference between the two systems was the representation. All results in this section are from the same set of twenty runs, ten using the direct representation and ten the generative representation.

4.1 Fitness Comparison

Our first graph, Figure 4, plots the average fitness (over 10 runs) of the best individuals evolved with the direct representation against the best evolved with the generative representation. The final fitness values achieved were:

Direct: 134, 744, 42, 48, 62, 74, 42, 66, 86, 312.
Generative: 8180, 664, 2308, 3386, 696, 224, 1880, 364, 3810, 4556.

After 10 generations the generative representation achieved a higher average fitness than runs with the direct representation did after 250 generations, and the final genobots evolved with the generative representation were more than 10 times faster, on average, than robots evolved with the direct representation.

Figure 5 shows the best two individuals evolved with the direct representation (a and b) and the best two evolved with the generative representation (c and d). From the images it can be seen that the robots evolved with the direct representation are irregular and have few components, whereas the robots evolved with the generative representation are more regular and, in some cases, have two or more levels of reused assemblies of components. Furthermore, the network morphologies constructed from the generative representation also contain some reuse of subnetworks. The neural network controller shown in Figure 6 is the controller for the genobot in Figure 5h. In

2 A different approach would be to put a limit on the maximum torque applied on a connection, but this would require a simulator with more detailed physics than the one used here.

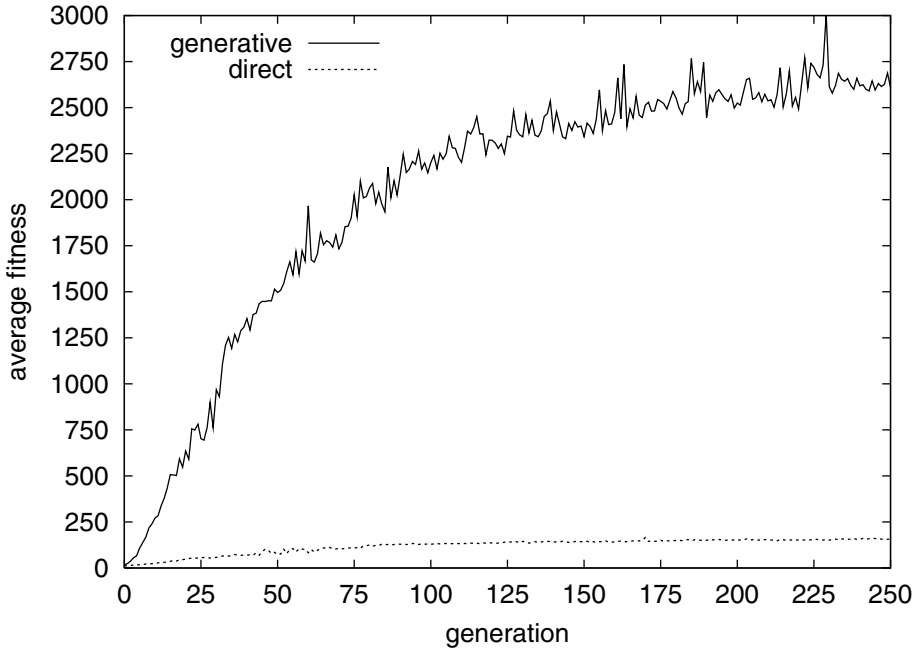


Figure 4. Fitness comparison between the direct representation and the generative representation.

addition to its reuse of components, its linear sequence of outputs corresponds to the linear sequence of joints in the genobot's morphology.

4.2 Reuse and Evolvability

In our introduction we argued that one advantage of a generative representation is its ability to create more complex components from simpler ones. The graph in Figure 7a shows, for each generation, the average length of the genotype for both representations, and the length of the assembly procedure produced by the generative representation. In the initial populations that used the generative representation the average length of the genotype was 126 symbols and the average length of the generated assembly procedure was 534 symbols. This means that on average each symbol in the genotype was being used 4.2 times in creating the assembly procedure. After 250 generations, this evolved to an average length of the genotype of 208 symbols and an average length of the resulting assembly procedure of 2,387 symbols, which is an average reuse of 11.5. The average number of parts (rods only) used in a design is plotted in the graph in Figure 7b. As the genotype sizes for direct and for generative representations are about the same, the increased number of parts used in designs constructed from the generative representation suggests that the multiple expression of genotype produced a reuse of parts. Further support for this assertion comes from the images in Figures 5c–h, which show that designs evolved with the generative representation have the same assemblies of parts occurring multiple times in a genobot.

The second part of our argument for a generative representation is that, through evolution, useful bias of the of search space becomes embedded in a design encoded with a generative representation, resulting in better performance of the variation operators. To compare the performance in variation operators between the two representations we compare the change in fitness between a parent and its child (from mutation) and

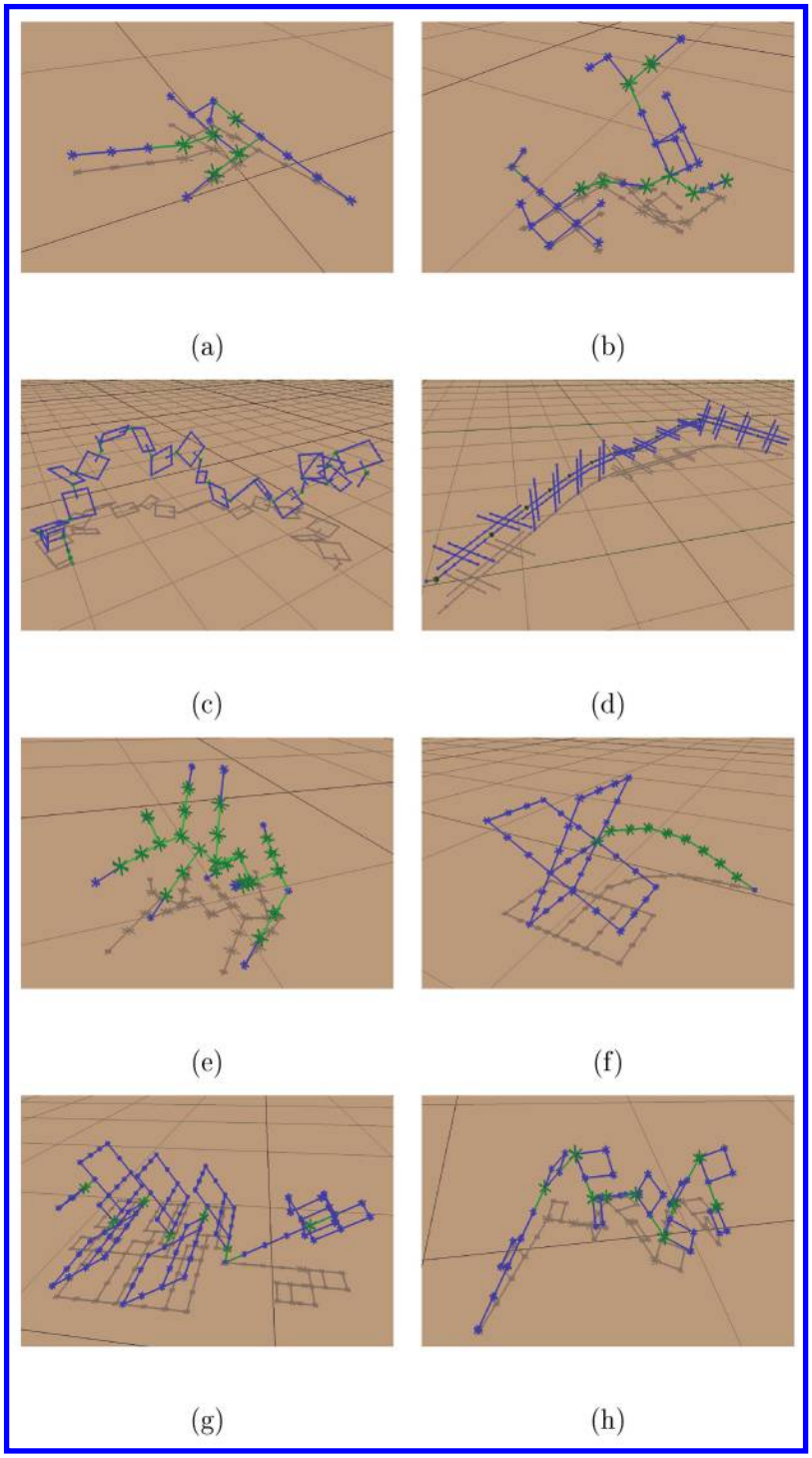


Figure 5. The best two individuals evolved with: (a) and (b), the direct representation; (c) and (d), the generative representation. Genobots (e)–(h) were evolved with the generative representation and no constraints on limb lengths.

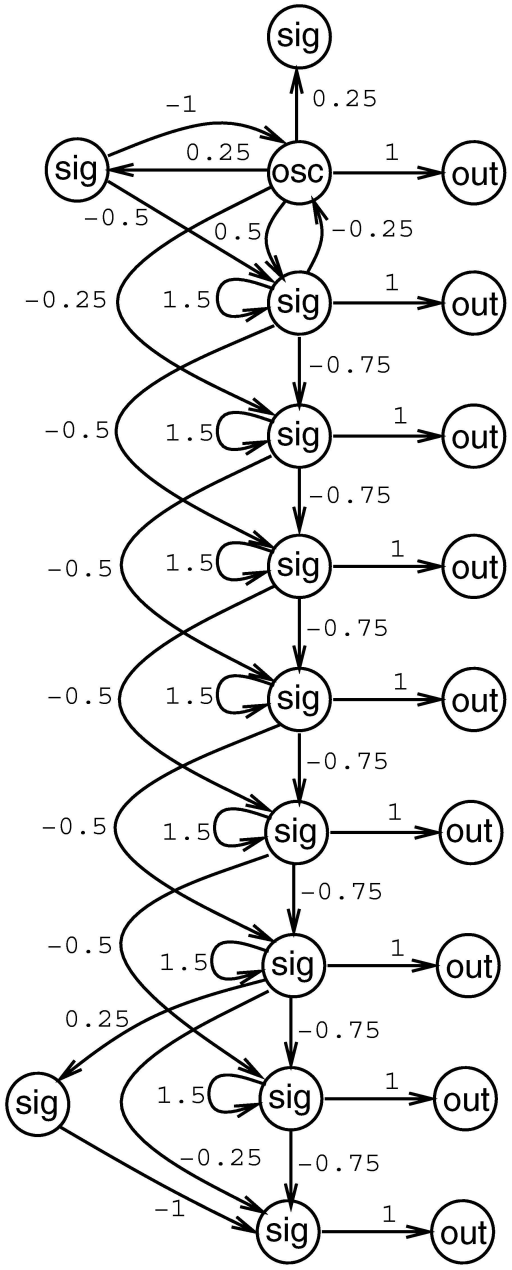
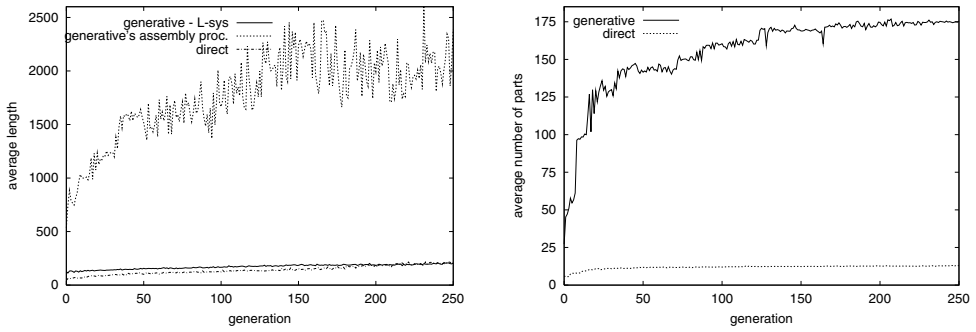


Figure 6. Evolved neural network controller for the genobot in Figure 5h.

plot it against the difference between parent's and child's assembly procedures. For the direct representation, the assembly procedure is the same as the genotype; for the generative representation, the assembly procedure is the last string produced by the L-system. In the case where the assembly procedures are the same length, the command difference between two assembly procedures is the number of locations for which the parent and child have different symbols. When strings have different lengths, the



(a) (b)

Figure 7. Graph of (a) length of the genotypes, and of the command string produced by the generative representation, against generation, and (b) number of parts against generation.

number of occurrences of each symbol is counted and the command difference is the sum of the differences between these values.[3]

Figure 8 shows four different plots of change in fitness against command difference. The first graph, Figure 8a, is for a single mutation operator applied to a direct representation. Most mutations were close to the parent, and most successful mutations were less than 10 commands apart. As offspring under the generative representation will tend to be further from their parent (because of reuse of the genotype), we also plot change in fitness against command difference for the direct representation with 1–6 mutations (chosen with uniform probability) applied. From the graphs it can be seen that mutations on the direct representation were usually successful only when the change in command difference between assembly procedures was small (less than 10), and even then improvements were not large. The graph in Figure 8c is a plot of change in fitness against command difference between assembly procedures and shows that with the generative representation there was a larger variation in assembly procedure distance between a parent and its child than with the direct representation. This graph also shows that offspring were more likely to have higher fitness than their parents with the generative representation than with the direct representation.

To determine if this improved performance under variation was only a result of the types of strings generated by the generative representation, we also applied 1-6 mutations to the assembly procedure produced by the generative representation. The plot of Figure 8d shows that variation of the generative representation's assembly procedure was not as successful as variation of the generative representation itself, suggesting that the structure with the generative representation had captured some useful bias of the design problem over the course of evolution.

As a way of normalizing for the higher average fitness achieved with the generative representation, we next show the rate of success (a child's fitness greater than its parent's) for the mutation operator for different command differences between parent and child (Figure 9). Again we include a comparison with one to six mutations applied to the direct representation as well as one to six mutations applied to the assembly procedure produced by the generative representation. With the direct representation, the success rate of the mutation operator quickly dropped to zero as the difference

3 An alternative distance metric that could be used is edit distance, but this is an expensive computation for bracketed strings (effectively trees) and would be prohibitive [31].

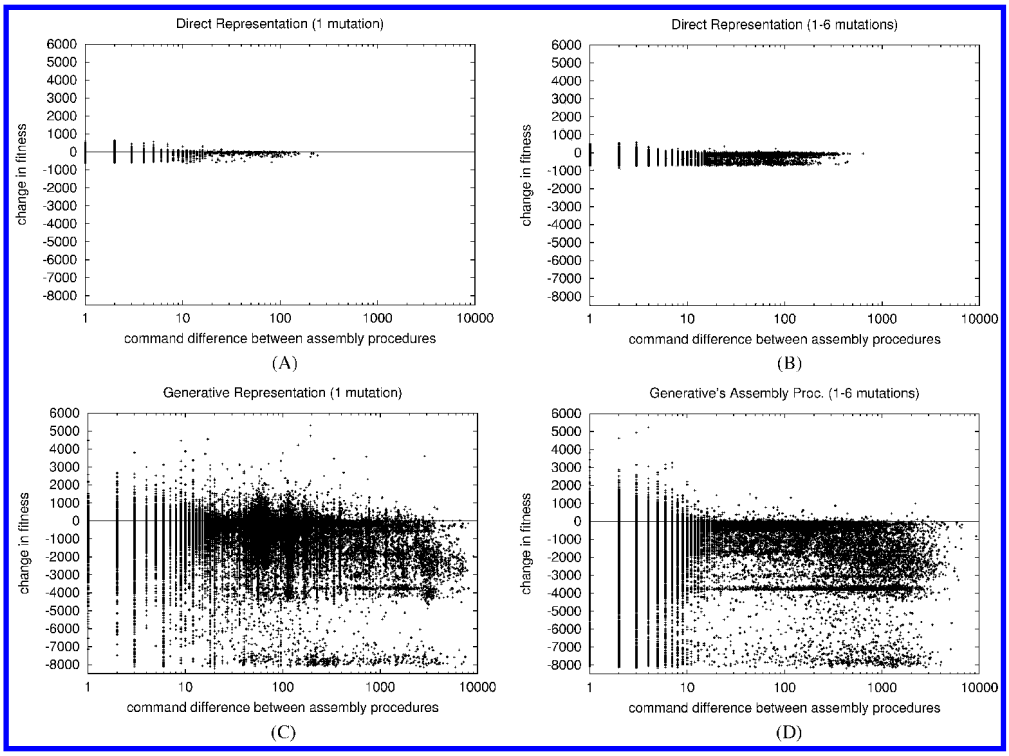


Figure 8. Plot of amount of change in genotype from parent to child against change in fitness.

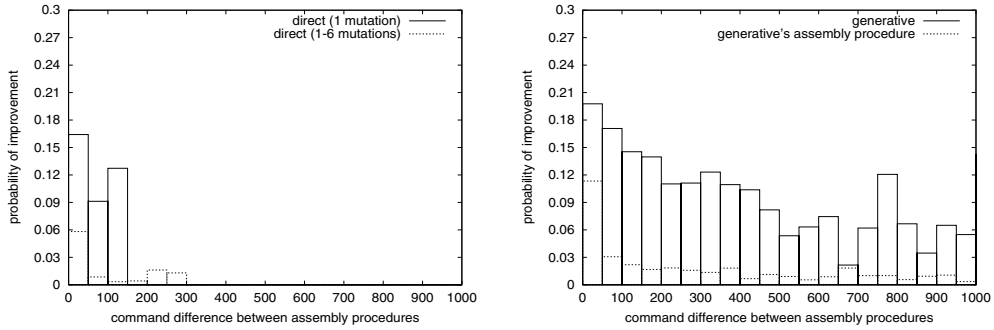


Figure 9. Probability of success (child is more fit than parent) comparison between different representations, for ranges 1–50, 51–100, 101–150, . . .

between parent's and child's assembly procedures increased. In contrast, the success rate of mutation decayed more gracefully with increasing command difference when the mutation was applied to the generative representation—even when parent and child were 500 construction symbols apart, the success rate was 10%. The higher success rate of mutation, especially with larger differences in assembly procedures, and greater average increase in fitness with the generative representation provide strong evidence that the generative representation has captured meaningful bias of the design problem.

Table 4. Summary of results for evolving neural-network-controlled robots.

	Value	
Quantity	Direct	Generative
Average final best fitness	157	2609
Number of mutations with distance > 0	184,667	188,749
Success rate of mutations	16.4%	18.1%
Average fitness change	−27	−526
Average fitness change of successful mutations	19	178
Average distance of mutations	3	118
Average distance of successful mutations	3	44
Number of large mutations (distance > 100)	65	27,899
Success rate of large mutations (distance > 100)	10.8%	10.5%
Average fitness change of large mutations (distance > 100)	−74	−1396
Average fitness change of successful large mutations (distance > 100)	25	357
Average distance of large mutations (distance > 100)	131	693
Average distance of successful large mutations (distance > 100)	113	344

4.3 Summary of Results

The results of our experiments are summarized in Table 4. While the overall success rate of mutations is similar between the two representations, there is a difference in the average distance of mutations. Mutations for the direct representation were considerably smaller in assembly-procedure command difference than for the generative representation. Examining the probability of success of a mutation for similar sized distances (the graphs in Figure 9, which plot the rate of success for distance in bins of size 50), it can be seen that for all distances the rate of success was higher with the generative representation. In addition, considering only mutations that are successful (the child has higher fitness than its parent), the average increase in fitness was significantly higher with the generative representation than with the direct representation.

5 Discussion

By allowing the inclusion of subprocedure-like structures (here, the L-system's production rules) a generative representation can create more complex building blocks from simpler ones. Since these production rules are a single character that can be inserted or removed from the genotype with a single mutation, variation operators can scale with design complexity because new assemblies of components become possible with unit variations. In addition, reusing code in the genotype to reuse parts in the actual design makes certain types of design changes easier. The images in Figure 10 show examples of reuse through variations applied to the individual of Figure 5d. Changing the genotype to add rods to an assembly of parts results in a change in all occurrences of that part in the design (Figure 10b), and a single change in the genotype can cause the addition or subtraction of a large number of parts (Figure 10c). In both cases the same change would be harder to make with a direct representation. For example, even though recombination can duplicate assemblies of parts in a direct representation, a

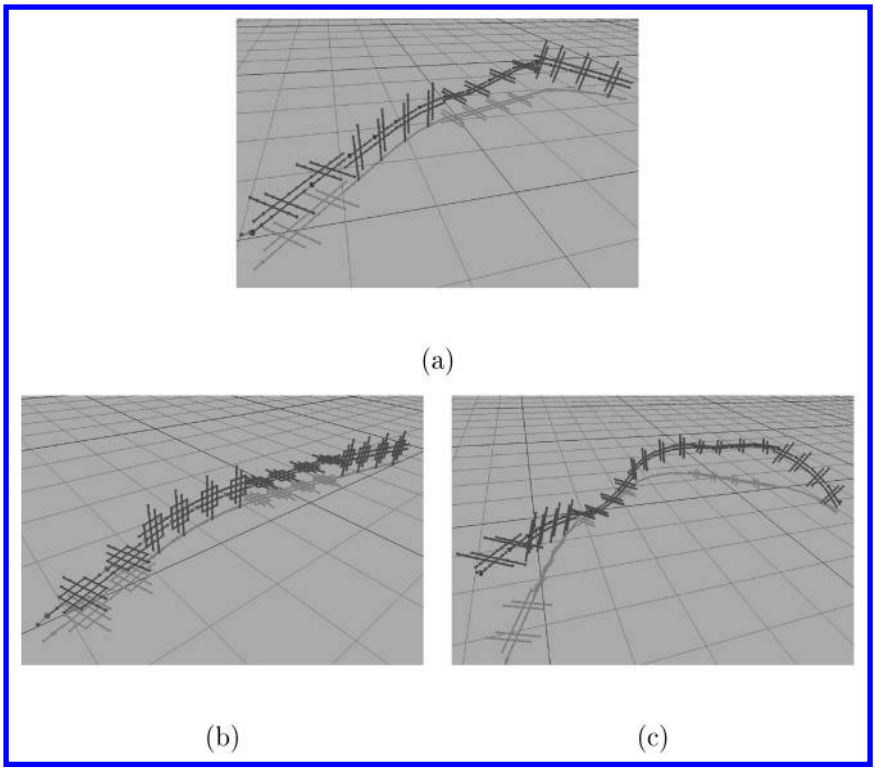


Figure 10. Mutations of a genobot: (a) the genobot from Figure 5d; (b) a change in a low-level component of parts causes all occurrences of this part to have the change; (c) a single change in the genotype changes the number of high-level components in the genobot from four to six.

later application of variation will only change one instance of this assembly. As designs become more complex, the possibility of the same change happening simultaneously in all uses of this assembly becomes increasingly unlikely with a direct representation, yet remains constant with a generative representation.

Of the representations described in Section 2, the generative representation of GENRE has similarities to those of Framsticks and the one used by Sims. The method of specifying a creature's morphology by a sequence of commands, with parentheses and brackets used for branching, is almost identical to the Framsticks *recur* representation [20]. One difference is in specifying the neural controller. In Framsticks, this is done by listing the links immediately after the neuron, whereas in our system a cellular encoding language is used. The other difference is that the Framsticks *recur* representation does not provide for reuse. Both Framsticks *devel* [21] and Sims' system [33] allow for parts of the genotype to be reused through a looping ability. This looping is like the repetition blocks of the generative representation defined in Section 3 and is the beginning of a procedural description. The representation of GENRE extends the ability to define loops by including labeled subprocedures with parameters in the genotype—similar to the modules of GLiB [4], the automatically defined functions (ADFs) of GP [22], and the automatically defined subnetworks (ADSNs) of [9]—and conditionals on the parameters.

In our comparison, robots evolved with a generative representation were, on average, ten times faster than those evolved with a direct representation. These results differ from those of [21], in which Komosinski and Rotaru-Varga's comparison produced little difference between a generative representation (*devel*), and a direct representation (*re-*

cur). Since their generative representation had only the property of iteration, whereas our generative representation also has conditionals and abstraction with parameters, this suggests that these additional properties can make a significant difference on the performance of an evolutionary design system.

6 Conclusion

The concurrent evolution of bodies and brains has been limited by the representations used to encode them [36, 20, 26, 6]. Here we have defined the class of generative representations and presented GENRE, a generic system for evolving designs with this class of representations. Previous work has shown how this system can be used to evolve table designs [14], two-dimensional oscillator-controlled genobots [12], and three-dimensional oscillator-controlled genobots [15]. Since GENRE treats command sets and assembly procedures as symbols and strings, this evolutionary system can be used, by replacing one command set with another and/or replacing the design constructor, on any design domain in which a design can be constructed from a linear assembly procedure.

In this paper we have described a method for evolving the morphology and neural controller of three-dimensional robots. We have shown that robots evolved with the generative representation reach higher fitness than those evolved with a direct representation. This improved performance has been shown to be the result of the ability of the generative representation to reuse parts of the design. In summary, generative representations accelerate evolution by learning useful problem bias over the course of the evolution and by encapsulating, in heritable elements of the genotype, assemblies of phenotypic components, thereby allowing mutation to scale with design complexity.

Generative representations capture the fundamental elements of general purpose programming languages—combination, control flow, and abstraction. However, we note that the linear representation is somewhat limiting, even though primitives like "push" and "pop" add tree-like constructions. As continuing work expands the range and power of generative representations while maintaining evolvability, we expect to see ever more progress toward general purpose evolutionary design.

Acknowledgments

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 (especially J. Rieffel, S. Viswanathan, and R. Watson), H. Lipson, M. Xiu, and the anonymous reviewers.

References

1. Abelson, H., & diSessa, A. A. (1982). *Turtle geometry.* Cambridge, MA: MIT Press.
2. Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). New York: McGraw-Hill.
3. Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., & Walter, P. (2002). *Molecular biology of the cell* (4th ed.). New York: Garland Publishing.
4. Angeline, P., & Pollack, J. B. (1994). Coevolving high-level representations. In C. Langton (Ed.), *Proceedings of the Third Workshop on Artificial Life.* Reading, MA: Addison-Wesley.
5. Boers, E. J. W., Kuiper, H., Happel, B. L. M., & Sprinkhuizen-Kuyper, I. G. (1993). Designing modular artificial neural networks. In H. A. Wijshoff (Ed.), *Proceedings of Computing Science in The Netherlands* (pp. 87–96). SION, Stichting Mathematisch Centrum.

6. Bongard, J. C., & Pfeifer, R. (2001). Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Genetic and Evolutionary Computation Conference* (pp. 829–836). San Francisco, CA: Morgan Kaufmann.

7. Coates, P., Broughton, T., & Jackson, H. (1999). Exploring three-dimensional design worlds using Lindenmayer systems and genetic programming. In P. J. Bentley (Ed.), *Evolutionary design by computers*.

8. Dawkins, R. (1986). *The blind watchmaker*. Harlow, UK: Longman.

9. Gruau, F. (1994). *Neural network synthesis using cellular encoding and the genetic algorithm*. Unpublished doctoral dissertation, Ecole Normale Supérieure de Lyon.

10. Grzeszczuk, R., & Terzopoulos, D. (1995). Automated learning of muscle-actuated locomotion through control abstraction. *Computer Graphics, 29*, 63–70.

11. Hornby, G. S., Fujita, M., Takamura, S., Yamamoto, T., & Hanagata, O. (1999). Autonomous evolution of gaits with the Sony quadruped robot. In *Proceedings of the Genetic and Evolutionary Computation Conference* (pp. 1297–1304). San Francisco, CA: Morgan Kaufmann.

12. Hornby, G. S., Lipson, H., & Pollack, J. B. (2001). Evolution of generative design systems for modular physical robots. In *IEEE International Conference on Robotics and Automation* (pp. 4146–4151). Piscataway, NJ: IEEE Press.

13. Hornby, G. S., Lipson, H., & Pollack, J. B. (2002). *Generative representations for the automatic design of modular physical robots* (Technical report). Waltham, MA: Computer Science Dept., Brandeis University.

14. Hornby, G. S., & Pollack, J. B. (2001). The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation* (pp. 600–607). Piscataway, NJ: IEEE Press.

15. Hornby, G. S., & Pollack, J. B. (2001). Evolving L-systems to generate virtual creatures. *Computers and Graphics, 25*(6), 1041–1048.

16. Hornby, G. S., Takamura, S., Hanagata, O., Fujita, M., & Pollack, J. (2000). Evolution of controllers from a high-level simulator to a high dof robot. In J. Miller (Ed.), *Evolvable systems: From biology to hardware, Proceedings of the Third International Conference (ICES 2000)*, Lecture Notes in Computer Science, Vol. 1801 (pp. 80–89). New York: Springer-Verlag.

17. Jacob, C. (1996). Evolution programs evolved. In H.-M. Voigt, W. Ebeling, I. Rechenberg, & H.-P. Schwefel (Eds.), *Parallel problem solving from nature, PPSN-IV*, Lecture Notes in Computer Science 1141 (pp. 42–51). Berlin: Springer-Verlag.

18. Jakobi, N. (1998). *Minimal simulations for evolutionary robotics*. Unpublished doctoral dissertation, School of Cognitive and Computing Sciences, University of Sussex.

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

20. Komosinski, M. (2000). The world of Framsticks: Simulation, evolution, interaction. In *Virtual Worlds 2*, Lecture Notes in Artificial Intelligence 1834 (pp. 214–224). New York: Springer-Verlag.

21. Komosinski, M., & Rotaru-Varga, A. (2001). Comparison of different genotype encodings for simulated 3D agents. *Artificial Life, 7*(4), 395–418.

22. Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

23. Koza, J. R. (1994). *Genetic programming II: Automatic discovery of reusable programs*. Cambridge, MA: MIT Press.

24. Lewin, B. (2000). *Genes VII*. Oxford, UK: Oxford University Press.

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

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

27. Luke, S., & Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. In J. Koza (Ed.), *Late-breaking papers of genetic programming 96* (pp. 117–124). Stanford, CA: Stanford Bookstore.

28. Ngo, J. T., & Marks, J. (1993). Spacetime constraints revisited. *Computer Graphics, 27*, 343–350.

29. Ochoa, G. (1998). On genetic algorithms and lindenmayer systems. In A. Eiben, T. Baeck, M. Schoenauer, & H. P. Schwefel (Eds.), *Parallel problem solving from nature V* (pp. 335–344). New York: Springer-Verlag.

30. Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants*. New York: Springer-Verlag.

31. Sankoff, D., & Kruskal, J. B. (Eds.). (1983). *Time warps, string edits and macromolecules: The theory and practice of sequence comparison*. Reading, MA: Addison-Wesley.

32. Sims, K. (1994). Evolving 3D morphology and behavior by competition. In R. Brooks and P. Maes (Eds.), *Proceedings of the Fourth Workshop on Artificial Life*, (pp. 28–39). Cambridge, MA: MIT Press.

33. Sims, K. (1994). Evolving virtual creatures. *Computer Graphics, 28*, 15–22.

34. Taylor, T., & Massey, C. (2001). Recent developments in the evolution of morphologies and controllers for physically simulated creatures. *Artificial Life,* 7(1), 77–87.

35. van de Panne, M., & Fiume, E. (1993). Sensor-actuator networks. *Computer Graphics, 27*, 335–342.

36. Ventrella, J. (1994). Explorations in the emergence of morphology and locomotion behavior in animated characters. In R. Brooks and P. Maes (Eds.), *Proceedings of the Fourth Workshop on Artificial Life*. Cambridge, MA: MIT Press.

37. Ventrella, J. (1999). Animated artificial life. In J.-C. Heudin (Ed.), *Virtual worlds: Synthetic universes, digital life, and complexity*. Reading, MA: Perseus Books.

This article has been cited by:

1. Lídio Mauro Lima de Campos. 2017. A neuro-evolutive algorithm biologically inspired based on rational agents. *International Journal of Hybrid Intelligent Systems* **9**, 1-14. [Crossref]
2. Lucas Helms, Jeff Clune. 2017. Improving HybrID: How to best combine indirect and direct encoding in evolutionary algorithms. *PLOS ONE* **12**:3, e0174635. [Crossref]
3. Lídio Mauro Lima de Campos, Roberto Célio Limão de Oliveira, Mauro Roisenberg. 2016. Optimization of neural networks through grammatical evolution and a genetic algorithm. *Expert Systems with Applications* **56**, 368-384. [Crossref]
4. . How Might Aging Have Evolved as an Adaptation? 165-191. [Crossref]
5. Justin K. Pugh, Lisa B. Soros, Kenneth O. Stanley. 2016. Quality Diversity: A New Frontier for Evolutionary Computation. *Frontiers in Robotics and AI* **3**. . [Crossref]
6. Hongjin Zhang, Yizuo Zhang, Weidong Zhang. A comparative study of optimization algorithms for correction of radar system error 3737-3742. [Crossref]
7. Gustavo A. Cardona, Wilfrido Moreno, Alfredo Weitzenfeld, Juan M. Calderon. Reduction of impact force in falling robots using variable stiffness 1-6. [Crossref]
8. Oveis Abedinia, Nima Amjady, Ali Ghasemi. 2016. A new metaheuristic algorithm based on shark smell optimization. *Complexity* **21**:5, 97. [Crossref]
9. . Evolutionary Programming and Heuristic Optimization 131-155. [Crossref]
10. Jamie Hewland, Geoff Nitschke. The Benefits of Adaptive Behavior and Morphology for Cooperation 1047-1054. [Crossref]
11. Khalid M. Salama, Ashraf M. Abdelbar. 2015. Learning neural network structures with ant colony algorithms. *Swarm Intelligence* **9**:4, 229-265. [Crossref]
12. James Watson, Geoff Nitschke. Evolving Robust Robot Team Morphologies for Collective Construction 1039-1046. [Crossref]
13. Paul Szerlip, Kenneth O. Stanley. 2015. Indirectly Encoding Running and Jumping Sodarace Creatures for Artificial Life. *Artificial Life* **21**:4, 432-444. [Abstract] [Full Text] [PDF] [PDF Plus]
14. O. Abedinia, N. Amjady, M. Shafie-khah, J.P.S. Catalão. 2015. Electricity price forecast using Combinatorial Neural Network trained by a new stochastic search method. *Energy Conversion and Management* **105**, 642-654. [Crossref]
15. Danesh Tarapore, Jean-Baptiste Mouret. 2015. Evolvability signatures of generative encodings: Beyond standard performance benchmarks. *Information Sciences* **313**, 43-61. [Crossref]
16. SAMEER GUPTA, EKTA SINGLA. 2015. Evolutionary robotics in two decades: A review. *Sadhana* **40**:4, 1169-1184. [Crossref]
17. James Watson, Geoff Nitschke. Deriving minimal sensory configurations for evolved cooperative robot teams 3065-3071. [Crossref]
18. Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, Agoston E. (Gusz) Eiben. 2015. Evolutionary Robotics: What, Why, and Where to. *Frontiers in Robotics and AI* **2**. . [Crossref]
19. Stephane Doncieux, Jean-Baptiste Mouret. 2014. Beyond black-box optimization: a review of selective pressures for evolutionary robotics. *Evolutionary Intelligence* **7**:2, 71-93. [Crossref]
20. René Doursat, Carlos Sánchez. 2014. Growing Fine-Grained Multicellular Robots. *Soft Robotics* **1**:2, 110-121. [Crossref]
21. Yao Yao, Kathleen Marchal, Yves Van de Peer. 2014. Improving the Adaptability of Simulated Evolutionary Swarm Robots in Dynamically Changing Environments. *PLoS ONE* **9**:3, e90695. [Crossref]
22. D. Lobo, M. Solano, G. A. Bubenik, M. Levin. 2014. A linear-encoding model explains the variability of the target morphology in regeneration. *Journal of The Royal Society Interface* **11**:92, 20130918-20130918. [Crossref]
23. Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, Julian F. Miller. 2013. Fast learning neural networks using Cartesian genetic programming. *Neurocomputing* **121**, 274-289. [Crossref]
24. Joel Lehman, Sebastian Risi, David D?Ambrosio, Kenneth O Stanley. 2013. Encouraging reactivity to create robust machines. *Adaptive Behavior* **21**:6, 484-500. [Crossref]
25. Paul Tonelli, Jean-Baptiste Mouret. 2013. On the Relationships between Generative Encodings, Regularity, and Learning Abilities when Evolving Plastic Artificial Neural Networks. *PLoS ONE* **8**:11, e79138. [Crossref]
26. David B. D'Ambrosio, Kenneth O. Stanley. 2013. Scalable multiagent learning through indirect encoding of policy geometry. *Evolutionary Intelligence* **6**:1, 1-26. [Crossref]

27. Krzysztof Krawiec, Tomasz Pawlak. 2013. Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators. *Genetic Programming and Evolvable Machines* **14**:1, 31-63. [Crossref]
28. Sebastian Risi, Kenneth O. Stanley. 2012. An Enhanced Hypercube-Based Encoding for Evolving the Placement, Density, and Connectivity of Neurons. *Artificial Life* **18**:4, 331-363. [Abstract] [Full Text] [PDF] [PDF Plus]
29. Sebastian Risi, Kenneth O. Stanley. A unified approach to evolving plasticity and neural geometry 1-8. [Crossref]
30. Jean-François Dupuis, Zhun Fan, Erik D. Goodman. 2012. Evolutionary Design of Both Topologies and Parameters of a Hybrid Dynamical System. *IEEE Transactions on Evolutionary Computation* **16**:3, 391-405. [Crossref]
31. J. Thangavelautham, G. M. T. D'Eleuterio. 2012. Tackling Learning Intractability Through Topological Organization and Regulation of Cortical Networks. *IEEE Transactions on Neural Networks and Learning Systems* **23**:4, 552-564. [Crossref]
32. J.-B. Mouret, S. Doncieux. 2012. Encouraging Behavioral Diversity in Evolutionary Robotics: An Empirical Study. *Evolutionary Computation* **20**:1, 91-133. [Abstract] [Full Text] [PDF] [PDF Plus]
33. Jimmy Secretan, Nicholas Beato, David B. D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T. Folsom-Kovarik, Kenneth O. Stanley. 2011. Picbreeder: A Case Study in Collaborative Evolutionary Exploration of Design Space. *Evolutionary Computation* **19**:3, 373-403. [Abstract] [PDF] [PDF Plus]
34. Jeff Clune, Kenneth O. Stanley, Robert T. Pennock, Charles Ofria. 2011. On the Performance of Indirect Encoding Across the Continuum of Regularity. *IEEE Transactions on Evolutionary Computation* **15**:3, 346-367. [Crossref]
35. Vinod K. Valsalam, Risto Miikkulainen. 2011. Evolving Symmetry for Modular System Design. *IEEE Transactions on Evolutionary Computation* **15**:3, 368-386. [Crossref]
36. Gregory. S. Hornby, Jason D. Lohn, Derek S. Linden. 2011. Computer-Automated Evolution of an X-Band Antenna for NASA's Space Technology 5 Mission. *Evolutionary Computation* **19**:1, 1-23. [Abstract] [PDF] [PDF Plus]
37. Yaochu Jin, Yan Meng. 2011. Morphogenetic Robotics: An Emerging New Field in Developmental Robotics. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **41**:2, 145-160. [Crossref]
38. J. Bongard. 2011. Morphological change in machines accelerates the evolution of robust behavior. *Proceedings of the National Academy of Sciences* **108**:4, 1234-1239. [Crossref]
39. Enrique Fernandez-Blanco, Julian Dorado, Jose A. Serantes, Daniel Rivero, Juan R. Rabuñal. Artificial Cells for Information Processing: Iris Classification 44-52. [Crossref]
40. Alessandro Fontana. Epigenetic Tracking: Biological Implications 10-17. [Crossref]
41. Daniel Lobo, Francisco J. Vico. 2010. Evolution of form and function in a model of differentiated multicellular organisms with gene regulatory networks. *Biosystems* **102**:2-3, 112-123. [Crossref]
42. Shuguang Li, Jianping Yuan, Franz Nigl, Hod Lipson. A cuboctahedron module for a reconfigurable robot 535-541. [Crossref]
43. Daniel Lobo, Francisco J. Vico. 2010. Evolutionary development of tensegrity structures. *Biosystems* **101**:3, 167-176. [Crossref]
44. Jason Gauci, Kenneth O. Stanley. 2010. Autonomous Evolution of Topographic Regularities in Artificial Neural Networks. *Neural Computation* **22**:7, 1860-1898. [Abstract] [Full Text] [PDF] [PDF Plus]
45. Maryam Mahsal Khan, Gul Muhammad Khan, Julian F. Miller. Evolution of neural networks using Cartesian Genetic Programming 1-8. [Crossref]
46. Ting Hu, Wolfgang Banzhaf. 2010. Evolvability and Speed of Evolutionary Algorithms in Light of Recent Developments in Biology. *Journal of Artificial Evolution and Applications* **2010**, 1-28. [Crossref]
47. Pierre De Loor, Kristen Manac'h, Jacques Tisseau. 2009. Enaction-Based Artificial Intelligence: Toward Co-evolution with Humans in the Loop. *Minds and Machines* **19**:3, 319-343. [Crossref]
48. J. Ruiz-del-Solar, R. Palma-Amestoy, R. Marchant, I. Parra-Tsunekawa, P. Zegers. 2009. Learning to fall: Designing low damage fall sequences for humanoid soccer robots. *Robotics and Autonomous Systems* **57**:8, 796-807. [Crossref]
49. Christopher MacLeod, Grant Maxwell, Sethuraman Muthuraman. 2009. Incremental growth in modular neural networks. *Engineering Applications of Artificial Intelligence* **22**:4-5, 660-666. [Crossref]
50. M. Mazzapioda, A. Cangelosi, S. Nolfi. Evolving morphology and control: A distributed approach 2217-2224. [Crossref]
51. Jean-Baptiste Mouret, Stephane Doncieux. Evolving modular neural-networks through exaptation 1570-1577. [Crossref]
52. Kenneth O. Stanley, David B. D'Ambrosio, Jason Gauci. 2009. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life* **15**:2, 185-212. [Abstract] [PDF] [PDF Plus]
53. Artur Matos, Reiji Suzuki, Takaya Arita. 2009. Heterochrony and Artificial Embryogeny: A Method for Analyzing Artificial Embryogenies Based on Developmental Dynamics. *Artificial Life* **15**:2, 131-160. [Abstract] [PDF] [PDF Plus]

54. Javier Ruiz-del-Solar, Rodrigo Palma-Amestoy, Paul Vallejos, R. Marchant, P. Zegers. Designing Fall Sequences That Minimize Robot Damage in Robot Soccer 271-283. [Crossref]
55. Wojciech Jaśkowski, Krzysztof Krawiec, Bartosz Wieloch. 2008. Multitask Visual Learning Using Genetic Programming. *Evolutionary Computation* **16**:4, 439-459. [Abstract] [PDF] [PDF Plus]
56. Jean-Baptiste Mouret, Stéphane Doncieux. 2008. MENNAG: a modular, regular and hierarchical encoding for neural-networks based on attribute grammars. *Evolutionary Intelligence* **1**:3, 187-207. [Crossref]
57. Jason D. Lohn, Gregory S. Hornby, Derek S. Linden. 2008. Human-competitive evolved antennas. *AI EDAM* **22**:03. . [Crossref]
58. Luisa Caldas. 2008. Generation of energy-efficient architecture solutions applying GENE_ARCH: An evolution-based generative design system. *Advanced Engineering Informatics* **22**:1, 59-70. [Crossref]
59. Kenneth O. Stanley. 2007. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines* **8**:2, 131-162. [Crossref]
60. J.B. Pollack. 2006. Mindless Intelligence. *IEEE Intelligent Systems* **21**:3, 50-56. [Crossref]
61. Nicolas Lassabe, Herve Luga, Yves Duthen. Evolving Creatures in Virtual Ecosystems 11-20. [Crossref]
62. K.O. Stanley, B.D. Bryant, R. Miikkulainen. 2005. Real-Time Neuroevolution in the NERO Video Game. *IEEE Transactions on Evolutionary Computation* **9**:6, 653-668. [Crossref]
63. J.C. Bongard, H. Lipson. 2005. Nonlinear System Identification Using Coevolution of Models and Tests. *IEEE Transactions on Evolutionary Computation* **9**:4, 361-384. [Crossref]
64. J. Teo, H.A. Abbass. 2005. Multiobjectivity and Complexity in Embodied Cognition. *IEEE Transactions on Evolutionary Computation* **9**:4, 337-360. [Crossref]
65. Sung Young Jung. 2005. A Topographical Method for the Development of Neural Networks for Artificial Brain Evolution. *Artificial Life* **11**:3, 293-316. [Abstract] [PDF] [PDF Plus]
66. Domenico Parisi. 2004. Internal robotics. *Connection Science* **16**:4, 325-338. [Crossref]
67. Raffaele Bianco, Stefano Nolfi. 2004. Toward open-ended evolutionary robotics: evolving elementary robotic units able to self-assemble and self-reproduce. *Connection Science* **16**:4, 227-248. [Crossref]
68. Gregory S Hornby. 2004. Functional Scalability through Generative Representations: The Evolution of Table Designs. *Environment and Planning B: Planning and Design* **31**:4, 569-587. [Crossref]
69. G.S. Hornby, H. Lipson, J.B. Pollack. 2003. Generative representations for the automated design of modular physical robots. *IEEE Transactions on Robotics and Automation* **19**:4, 703-719. [Crossref]
70. Alon Gal, Gady Mahal, Moshe Sipper. 2003. Evolutionary Plantographics. *Artificial Life* **9**:2, 191-205. [Abstract] [PDF] [PDF Plus]
71. R. Kicinger, T. Arciszewski, K. De Jong. Morphogenesis and structural design: cellular automata representations of steel structures in tall buildings 411-418. [Crossref]