Generative Representations for Evolutionary Design Automation

A Dissertation

Presented to

The Faculty of the Graduate School of Arts and Sciences

Brandeis University

Department of Computer Science

Jordan B. Pollack, Advisor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

Gregory S. Hornby

February, 2003

This dissertation, directed and approved by Gregory S. Hornby's Committee, has been accepted and approved by the Graduate Faculty of Brandeis University
in partial fulfillment of the requirements for the degree of:

DOCTOR OF PHILOSOPHY

Dean of Arts and Sciences

Dissertation Committee:

Jordan B. Pollack, Chair

Timothy J. Hickey

Mitch Cherniack

Peter J. Bentley

Acknowledgments

There were many people who helped me out in the last six years. Obviously I couldn't have done this without an advisor and, in addition to this, I'd like to thank Jordan for the many useful comments he had, especially on how to spin my papers. All the members of the DEMO lab (1996-2002) contributed in some way to my thesis, but there are a couple of people I'd like to single out. First, Hod Lipson helped me figure out how to use the 3D printer, he built my first two robots, and then was of invaluable help through the questions he asked and the data he wanted when we wrote our two papers together. Our journal paper became the framework for supporting the claims of this dissertation. Next, Richard Watson, Shiva Viswanathan and John Rieffel (who helped build my third robot) gave me very useful feedback in reading over my papers and thesis, and everyone in the group had something to say in my group presentations. I'd like to thank my committee for the comments the gave me on my dissertation and for their time, and I'd like to thank Mei Xiu for the biology references. Finally, I want to thank my parents, family and friends for their love and support.

ABSTRACT

Generative Representations for Evolutionary Design Automation

A dissertation presented to the Faculty of
the Graduate School of Arts and Sciences of
Brandeis University, Waltham, Massachusetts

by Gregory S. Hornby

In this thesis the class of generative representations is defined and it is shown that this class of representations improves the scalability of evolutionary design systems by automatically learning inductive bias of the design problem thereby capturing design dependencies and better enabling search of large design spaces. First, properties of representations are identified as: combination, control-flow, and abstraction. Using these properties, representations are classified as *non-generative*, or *generative*. Whereas non-generative representations use elements of encoded artifacts at most once in translation from encoding to actual artifact, generative representations have the ability to reuse parts of the data structure for encoding artifacts through control-flow (using iteration) and/or abstraction (using labeled procedures). Unlike non-generative representations, which do not scale with design complexity because they cannot capture design dependencies in their structure, it is argued that evolution with generative representations can better scale with design complexity because of their ability to hierarchically create assemblies of modules for reuse, thereby enabling better search of large design spaces. Second, *GENRE*, an evolutionary design system using a generative representation, is described. Using this system, a non-generative and a generative representation are compared on four classes of designs: three-dimensional static structures constructed from voxels; neural networks; actuated robots controlled by oscillator networks; and neural network controlled robots. Results from evolving designs in these substrates show that the evolutionary design system is capable of finding solutions of higher fitness with the generative representation than with the non-generative representation. This improved performance is shown to be a result of the generative representation's ability to

capture intrinsic properties of the search space and its ability to reuse parts of the encoding in constructing designs. By capturing design dependencies in its structure, variation operators are more likely to be successful with a generative representation than with a non-generative representation. Second, reuse of data elements in encoded designs improves the ability of an evolutionary algorithm to search large design spaces.

Contents

Acknowledgments vii

Abstract ix

List of Tables xv

List of Figures xvii

1 Introduction 1
- 1.1 Evolutionary Algorithms . . . 2
- 1.2 Design Representations . . . 4
- 1.3 Limitations of Non-Generative Representations . . . 7
- 1.4 Advantages of Generative Representations . . . 8
- 1.5 Contributions . . . 10
- 1.6 Outline . . . 12

2 Related Work 15
- 2.1 Evolution of Images . . . 17
- 2.2 Evolution of Shape . . . 19
  - 2.2.1 Non-Generative Representations for Shape . . . 19
  - 2.2.2 Implicit, Generative Representations . . . 22
  - 2.2.3 Explicit, Generative Representations . . . 24
  - 2.2.4 Summary . . . 26
- 2.3 Evolution of Neural Networks . . . 27
- 2.4 Concurrent Evolution of Morphology and Controller . . . 30
- 2.5 Classifications of Design Representations . . . 32
- 2.6 Summary . . . 36

3 Methods 39
- 3.1 Generative Representations . . . 40
  - 3.1.1 Basic L-systems . . . 40
  - 3.1.2 Turtle Interpretation and Bracketed L-systems . . . 41
  - 3.1.3 Parametric L-systems . . . 43
  - 3.1.4 L-systems as a Generative Representation . . . 46
  - 3.1.5 L-systems as a Non-Generative Representation . . . 47

3.2 Evolutionary Algorithm . . . 49
3.2.1 Initialization . . . 50
3.2.2 Mutation . . . 50
3.2.3 Recombination . . . 53
3.3 Summary . . . 54

4 Design Domains **55**
4.1 Voxel Structures . . . 55
4.1.1 Generative Representation Example for Voxel Structures . . . 56
4.2 Neural Networks . . . 58
4.3 Oscillator Controlled Robots . . . 63
4.4 Neural-Network Controlled Robots . . . 67
4.4.1 Generative Representation Example for Neural Network Controlled Genobots . . . 70
4.5 Robot Simulator . . . 71
4.6 Summary . . . 72

5 Results **73**
5.1 Tables . . . 74
5.1.1 Evolved Tables and Fitness Comparison . . . 75
5.1.2 Reuse on the Table Design Problem . . . 79
5.1.3 Summary of Results for the Table Design Problem . . . 83
5.2 Parity Solving Neural Networks . . . 84
5.2.1 Fitness Comparison for 7-Parity Networks . . . 85
5.2.2 Reuse for the 7-Parity Networks . . . 89
5.2.3 Summary of Results for the Parity Networks . . . 89
5.3 Oscillator Controlled Robots . . . 90
5.3.1 Evolved Oscillator-Controlled Robots and Fitness Comparison . . . 91
5.3.2 Reuse with Oscillator-Controlled Robots . . . 97
5.3.3 Summary of Results for Oscillator-Controlled Robots . . . 98
5.4 Neural Network Controlled Robots . . . 98
5.4.1 Evolved Neural Network-Controlled Robots and Fitness Comparison . . . 99
5.4.2 Reuse for the Network-Controlled Robots . . . 104
5.4.3 Summary of Results for the Neural Network-Controlled Robots . . . 106
5.5 From Design to Implementation . . . 106
5.6 Summary of Results . . . 106

6 Discussion **111**
6.1 Evolvability . . . 117
6.1.1 Evolvability of Table Designs . . . 120
6.1.2 Evolvability of Parity Networks . . . 122
6.1.3 Evolvability of Oscillator-controlled Robot Designs . . . 126
6.1.4 Evolvability of Neural-network-controlled Robot Designs . . . 129
6.1.5 Summary . . . 134
6.2 Searching Large Design Spaces . . . 135

6.3 Summary . . . 139

7 Conclusions **141**
7.1 Summary . . . 141
7.2 Future Work . . . 142

A An Evolved Table **149**

B A Neural-Network for 3/5/7-Parity **159**

C Sorting Programs **163**
C.1 Sorting Program Construction Language . . . 163
C.2 Sorting Program Results . . . 164
C.2.1 Fitness for Sorting Programs . . . 164
C.2.2 Reuse and Evolvability for the Sorting Programs . . . 164
C.3 Summary of Results for the Sorting Programs . . . 166

D Two-dimensional Robots **167**
D.1 Two-dimensional Robot Construction Language . . . 167
D.2 Generative Representation Example for Oscillator Controlled Genobots . . . 169
D.3 Evolved Two-dimensional Genobots . . . 171

E An Oscillator-Controlled Genobot **173**

F A Neural-Network Controlled Genobot **177**

G Source Code **181**
G.1 ind_lsys.hh . . . 182
G.2 ind_lsys.cc . . . 186
G.3 prod_body . . . 199
G.4 prod_body.hh . . . 199
G.5 prod_body.cc . . . 203

List of Tables

2.1 Properties of the different representations for evolving images. 19
2.2 Properties of the different representations for the evolution of shapes. 26
2.3 Properties of the different representations for constructing neural networks. . 29
2.4 Properties of the different representations. 32

3.1 Design language for voxel structures. 42
3.2 Properties of *GENRE*'s non-generative and generative representations. 54

4.1 Design language for voxel structures. 56
4.2 Design language for neural networks. 60
4.3 Weight matrix. 62
4.4 Sequence of activation values. 63
4.5 Design language for oscillator-controlled, three-dimensional robots. 65
4.6 Design language for neural-network controlled, three-dimensional robots. . . . 68

5.1 Summary of results for 7-parity networks. 90

6.1 Average fitness change of successful mutations. 134
6.2 Average command difference of successful mutations. 134

C.1 List of commands for sorting programs. 163
C.2 Summary of results for sorting programs. 165

D.1 Design language for oscillator-controlled, two-dimensional robots. 167

List of Figures

1.1 The canonical evolutionary algorithm. 2
1.2 Parameterization of a table. 5
1.3 Three types of design representations. 6
1.4 Classes of design representations. 7
1.5 Two changes to the generative representation in figure 1.3.c. 9
1.6 Three classes of designs: (a), voxel structures; (b), neural networks; and (c), robots. 11

3.1 Drawing with a turtle. 41
3.2 Drawing with a turtle, using brackets. 42
3.3 Trees constructed from the L-system in section 3.1.2. 44
3.4 Graphical rendition of the generative representation, (a), along with the sequence of strings produced, (b). 48
3.5 A single mutation to the individual in figure 3.4 produces the L-system in (a), which produces the sequence of strings in (b). 52

4.1 Building an object. 57
4.2 Two example structures. 59
4.3 Construction of a neural network. 61
4.4 Basic building blocks of the system: rods of regular length and fixed and actuated joints. 64
4.5 Building and simulating a three-dimensional robot. 66
4.6 Constructing a neural-network controlled robot 69

5.1 Fitness comparison between the non-generative and generative representations on evolving tables, averaged over fifty trials. 75
5.2 The best six tables evolved using the non-generative representation. 76
5.3 The best six tables evolved using the generative representation. 77
5.4 Graphs of: (a) height; (b) penalty; (c) stability; and (d) surface values against generation on the table design problem. 78
5.5 Other tables evolved using the non-generative representation. 80
5.6 Other tables evolved using the generative representation. 81
5.7 More tables evolved using the generative representation. 82
5.8 Graph of (a) length of the genotypes, and assembly procedure produced by the generative representation, against generation; and (b) graph of number of parts against generation. 83

5.9 Fitness comparison between the non-generative and generative representations on evolving 7-parity networks. 85
5.10 The five correct parity networks evolved with the non-generative representation. 86
5.11 The first six correct parity networks evolved with the generative representation. 87
5.12 The second six correct parity networks evolved with the generative representation. 88
5.13 Graph of (a) length of the genotypes, and assembly procedure produced by the generative representation, against generation; and (b) graph of number of parts against generation. 89
5.14 Performance comparison between the non-generative and generative representations on evolving robots with oscillator networks for controllers. 91
5.15 The best six oscillator controlled robots evolved using the non-generative representation. 93
5.16 The best six oscillator controlled genobots evolved using the generative representation. 94
5.17 Part of the locomotion cycle of an oscillator-controlled genobot. 95
5.18 A variety of evolved 3D oscillator robots. 96
5.19 Graph of (a) length of the genotypes, and command string produced by the generative representation, against generation, and (b) graph of number of parts against generation. 97
5.20 Performance comparison between the non-generative representation and the generative representation on evolving robots with neural networks for controllers. 100
5.21 The best six neural-network controlled robots evolved with the non-generative representation. 101
5.22 The best six neural-network controlled genobots evolved with the generative representation. 102
5.23 Other genobots evolved using the generative representation on runs with no constraints on limb lengths. 103
5.24 Graph of (a) length of the genotypes, and command string produced by the generative representation, against generation, and (b) graph of number of parts against generation. 103
5.25 Evolved neural network controller for the genobot in figure 5.23.d. 105
5.26 Evolved tables shown both in simulation (left) and reality (right). 107
5.27 Evolved robots shown both in simulation (left) and reality (right). 108

6.1 Mutations of a table. 112
6.2 Mutations of a genobot: (a), the genobot from figure 5.22.b; (b), a change to a low-level component of parts results in all occurrences of this part to have the change; (c), a single change to the genotype changes the number of high-level components in the genobot from four to six. 113
6.3 Evolution of a four-legged walking genobot. 115
6.4 Evolution of a rolling genobot. 116

6.5 Tables: plot of amount of change in genotype from parent to child versus change in fitness (one out of every four data points) for cases with positive change. 119
6.6 Table designs: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d). 121
6.7 Tables: probability of improvement (child is more fit than parent) comparison between non-generative and generative representations, for ranges 1-50, 51-100, 101-150, ... 1951-2000. 122
6.8 7-Parity: plot of command-differences/edit-distances between parent and child's assembly procedures versus change in fitness (positive improvements only). . . 123
6.9 7-Parity: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d). 124
6.10 7-Parity: probability of improvement (child is more fit than parent) comparison between non-generative and generative representations (both command difference and edit distance) for ranges 1-10, 21-30, 31-40, ... 191-200. 125
6.11 Oscillator robots: plot of amount of change in assembly procedures from parent to child versus change in fitness. 127
6.12 Oscillator-controlled robots: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d). . . 128
6.13 Oscillator-controlled robots: probability of improvement (child is more fit than parent) comparison between the non-generative and generative representations, for ranges 1-50, 51-100, 101-150, ... 951-1000. 129
6.14 Plot of amount of change in genotype from parent to child against change in fitness. 130
6.15 Neural network controlled robots: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d). 132
6.16 Neural network controlled robots: probability of improvement (child is more fit than parent) comparison between different representations, for ranges 1-50, 51-100, 101-150, ... 951-1000. 133
6.17 Plots of number of parts versus fitness of the best individual from each trial: (a) and (b) are for tables (plotting one out of every ten points); (c) and (d) are for parity-solving neural networks; (e) and (f) are for 3D, oscillator controlled robots; and (g) and (h) are for 3D, neural-network controlled robots. 136
6.18 Plots of genotype length versus fitness of the best individual of each generation from each trial: (a) and (b), table designs (one out of every ten points); (c) and (d) are for parity-solving neural networks; (e) and (f), 3D, oscillator controlled robots; and (g) and (h), 3D, neural-network controlled robots. . . . 137

6.19 Graph of length of the genotypes and assembly procedure produced by the generative representation against generation for: (a) table designs; (b) parity networks; (c) oscillator controlled robots; and (d) neural-network controlled robots. . . . 138

7.1 A graphical rendition of: (a), a tree-structured generative representation; and (b), the tree-structured assembly procedure it produces. . . . 143
7.2 Four different tables constructed using different parameter values with the same design encoding. . . . 145
7.3 Networks constructed to solve 3, 5, 7-parity from the same evolved network encoding. . . . 146

A.1 Stages in the development of a table. . . . 152

C.1 Performance for the evolution of sorting programs using the generative representation. . . . 165
C.2 Graph of length of the generative representation genotype and the assembly procedure it produced. . . . 166

D.1 A sample oscillator-controlled, two-dimensional genobot. . . . 168
D.2 The locomotion cycle of a walking star creature, built from 43 bars and 24 actuated joints. . . . 170
D.3 Evolved creatures: *a*, a multi-legged walker with 24 bars and 12 actuated joints; *b*, a rolling circle with 23 bars and 6 actuated joints; *c*, an inch-worm built from 143 bars and 16 actuated joints; and *d*, an undullating serpent with 164 bars and 61 actuated joints; Notice the re-use of components. . . . 172

Chapter 1

Introduction

Evolutionary algorithms are a family of population-based search algorithms that have been used as systems for the design of antennas [85], flywheels, load cells [105], trusses [93], simulated robots [124], and other structures [5]. While they have been successful at producing simple, albeit novel artifacts, a concern with these systems is how well their search ability will scale to the larger design spaces associated with more complex artifacts [32] [103]. In engineering and software development, complex artifacts are achieved by exploiting the principles of regularity, modularity, hierarchy and reuse [129] [64] [92]. These features can be summarized as the hierarchical reuse of organizational units. In this dissertation *generative representations* are defined as the class of representations which allow reuse of encoded data elements, and it is shown that they improve the scalability of evolutionary design systems by capturing useful bias of the design problem and better enabling search of large design spaces.

The rest of this chapter is organized as follows. First, section 1.1 introduces evolutionary algorithms. Next, in section 1.2 the reason for focusing on representations as the means to improve the scalability of automated design systems is discussed, and generative, and other types of design representations, are defined. In section 1.3 it is argued that non-generative representations will not scale to complex artifacts, and this is followed by arguments for the improved scalability of generative representations, section 1.4. The final two sections list

the contributions of this dissertation, section 1.5, and an outline of the following chapters, section 1.6.

1.1 Evolutionary Algorithms

An evolutionary design system uses an evolutionary algorithm (EA) to search a design space. EAs are a family of population-based, stochastic search and optimization techniques inspired by natural evolution, which include genetic algorithms [55], evolutionary strategies [11], genetic programming [81] and evolutionary programming [39]. An EA operates by repeatedly processing a population of candidate solutions, called *individuals*, or *strings*. Search begins by creating an initial population of individuals, then iteratively selecting good individuals to reproduce, creating and evaluating new individuals, and inserting the new individuals into the population figure 1.1. A single cycle of selection-variation-evaluation-replacement is called a *generation*. The rest of this section describes each of these phases.

```
Initialize the population
Evaluate all members of the population
while not done {
        Select individual(s) in the population to be parent(s)
        Create new individuals by applying the variation operators to the parents
        Evaluate the new individuals
        Replace some/all of the individuals in the population with the new
        individuals
}
```

Figure 1.1: The canonical evolutionary algorithm.

Before evolution begins, an initial population of individuals is created randomly. Random solutions to the problem are created by assigning random values to each data element of an individual, called a *gene*. These values are an individual's *genotype* and the solution it maps to is the individual's *phenotype*. Using the binary representation for encoding numbers as an example, `010110` is the individual's *genotype* for which the value 22 is the individual's *phenotype*.

The evaluation of candidate solutions is common to all computer-based search algorithms, and is created specifically for each problem. In the EA community this evaluation function is called the *fitness function* and, in conjunction with the selection method, controls the number of offspring created by an individual. Using the fitness scores assigned by the fitness function, the selection phases chooses some subset of the current population as parents to create new individuals. By biasing selection toward individuals with better fitness, the created offspring are more likely to have higher fitness. Depending on how individuals are chosen, the population will quickly focus on promising individuals and converge quickly, or will maintain a diverse population and explore the domain.

Once the set of parent individuals has been selected, new individuals are created through variation of these existing solutions. The two methods of applying variation are *mutation* and *recombination*. Mutation acts on a single individual and is similar to the variation operator used in hill climbing or simulated annealing. Unlike search algorithms that operate on only a single candidate solution, EAs maintain a population of individuals which allows for creating new candidate solutions from multiple existing individuals. Recombination is a variation operator that combines parts of two individuals to create new ones.

In addition to controlling the search through selection, search can also be directed by the replacement method. Some EAs replace the entire population with the newly generated offspring. Others keep some of the better individuals and replace the rest with the newly created individuals.

As with other search algorithms, evolutionary search is run until a prespecified stopping criteria is met. The most common stopping criteria is to run the EA for a predetermined number of generations. If the fitness value of the desired solution is known ahead of time, search can be run until a solution with this fitness is found. Alternatively, the EA can be run until the population has converged. Convergence can be a measurement of the diversity of the population – in which case the search halts after the population diversity is below some predetermined value – or failure to find a new best point after a given number of generations.

One of the distinguishing characterics between the different branches of evolutionary algorithms is the representation upon which the operate. Genetic algorithms (GAs), developed by Holland at the University of Michigan, [55], distinguish between the genotype and the phenotype. With GAs the genotype, which is what the variation operators act upon, is usually encoded as a binary string and is translated to the phenotype for evaluation. Independent of the work of Holland, evolutionary strategies (ESs) were developed in Germany in the 1960s by Bienert, Rechenberg and Schwefel [11] and in California evolutionary programming (EP) was developed by Fogel, Owens and Walsh [39]. Both ESs and EP operate on individuals encoded as a real-valued vector – although EP started as the evolution of finite state machines before changing to real-valued vectors [38]. Finally, genetic programming (GP) is Koza's extension of GAs in which the individual being evolved is a computer program [81]. Although, unlike GAs, the mapping from genotype to phenotype is discarded and genotype is a computer program.

1.2 Design Representations

To justify investigation into design representations, it is necessary to examine the different parts of an evolutionary design system to see where scalability, through the hierarchical reuse of organizational units, can be achieved. Breaking down an evolutionary design system into its separate modules yields the search algorithm, the representation and the design problem [71]. Assuming that the problem has already been selected, that leaves only the search algorithm and the design representation to be considered. While the evolutionary algorithm can affect the degree of reuse in an evolved design, the ability to create structures which reuse subassemblies is limited by the ability of the representation to encode them. For example, with the parameterization of a table shown in figure 1.2, no modification to the evolutionary algorithm can affect the degree of reuse in an evolved design, nor is the hierarchical construction of organizational units possible. Thus the ability to automatically generate structures which have a reuse of subassemblies is strongly dependent on the representation

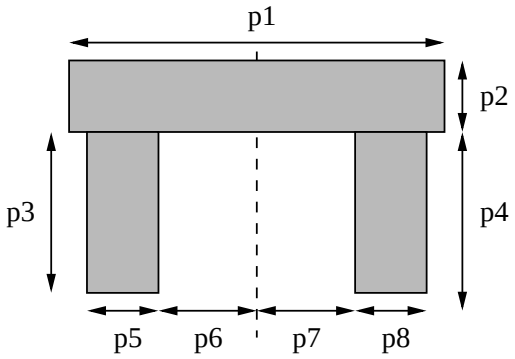
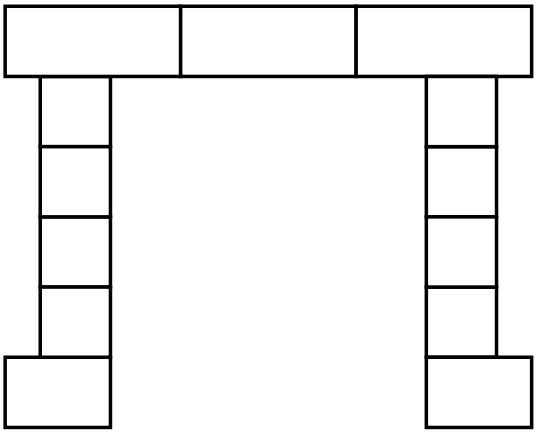


Figure 1.2: Parameterization of a table.

used by the evolutionary design system.

The different types of representations for evolutionary designs systems can be classified by how they encode designs. First, designs can be split into *parameterizations*, such as the parameterization of a table in figure 1.2, or *open-ended representations*. Since one of the goals of evolutionary design systems is to achieve truly novel artifacts, this dissertation focuses on open-ended representations, those in which the topology of a design is changeable, because it is difficult for a parameterization to achieve a type of design that was not conceived of by its programmers. Open-ended representations can be split into *non-generative* and *generative* representations. A **generative representation** is one in which an encoded design can reuse elements of its encoding in the translation to an actual design, whereas with a non-generative representation each element of an encoded design is used at most once in mapping to an actual design. Both non-generative and generative representations can be further split into two subcategories. The two subcategories of non-generative representations are *direct* and *indirect* representations. With a direct representation, the encoded design is essentially the same as the actual design (figure 1,3.a), and with an indirect representation there is a translation, or construction process, in going from the encoding to the actual design (figure 1.3.b). The two subcategories of generative representations are *implicit* and *explicit*. While both generative paradigms are indirect, implicit representations can be thought of as a direct representation with reuse, and explicit representations can be thought of as



(a) Direct mapping (a non-generative representation).

```
put-brick3
turn-up
put-brick1
put-brick1
put-brick1
put-brick1
turn-down
put-brick2
put-brick2
put-brick2
turn-down
put-brick1
put-brick1
put-brick1
put-brick1
put-brick3
```

(b) Indirect mapping (a non-generative representation).

```
build-leg()
(
    repeat (4 times) (
        put-brick1
    )
)

build-surface(width) (
    repeat (width times) (
        put-brick2
    )
)

main
(
    put-brick3
    turn-up
    build-leg()
    turn-down
    build-surface(3)
    turn-down
    build-leg()
    put-brick3
)
```

(c) Explicit, generative representation.

Figure 1.3: Three types of design representations.

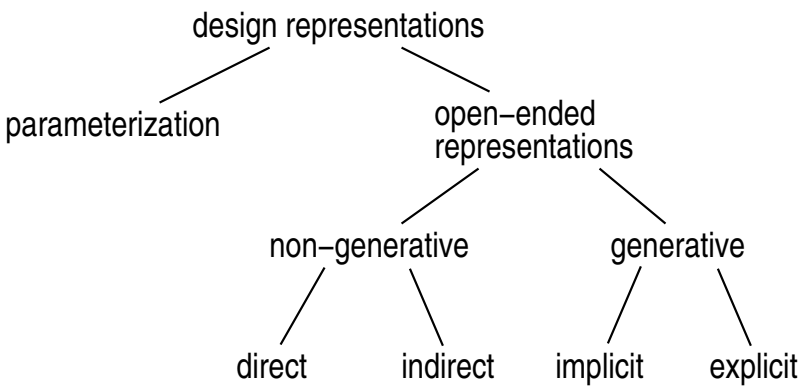


Figure 1.4: Classes of design representations.

indirect representations with reuse. Implicit, generative representations consist of a set of rules that implicitly specify a shape, such as through an iterative construction process similar to a cellular automata (CA). Explicit, generative representations are a procedural approach in which a design is explicitly represented by an algorithm for constructing it (figure 1.3.c). Thus explicit generative representations are like GP in that the genotype is a kind of computer program but, unlike GP and like GAs, this genotype is compiled into the phenotype. This hierarchy of design representations is shown in figure 1.4.

1.3 Limitations of Non-Generative Representations

For a number of years it has been recognized that representing designs either directly or indirectly with non-generative representations will not scale to complex structures [32] because of the exponential growth in the size of the design space and because the increasing number of dependencies in a design make it more difficult to make changes to a design. In the first case, as a design grows in the number of parts the expected distance (in number of parts) between a starting design and the desired optimized design increases. Conversely, changing a single part makes a proportionately smaller and smaller move towards the desired design. One consequence of this is that as designs increase in the number of parts, search algorithms will require more steps to find a good solution. Increasing the size of variation

(by changing more parts at a time) is not a solution because as the amount of variation is increased, the probability of the variation being advantageous decreases. The second case is similar: as designs become more complex, dependencies develop between parts of a design such that changing a property of one part requires the simultaneous change in another part of the design. For example, if the diameter of a screw is changed, then the diameter of the corresponding nut must also be changed or the parts will no longer fit together. Similarly, if the length of a table leg is changed, then all of the other table legs must be changed or the table will be unbalanced. Non-generative representations are not well suited to handling these increases in size and complexity because their language for representing designs is static.

1.4 Advantages of Generative Representations

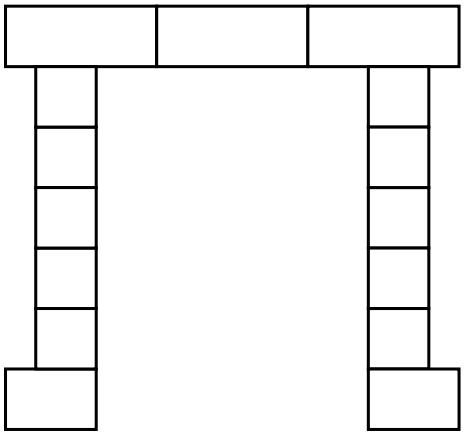
Unlike a non-generative representation, the ability to reuse elements of an encoded design improves the ability of search to navigate large design spaces and improves scalability by capturing design dependencies. First, navigation of large design spaces is improved through the ability to manipulate assemblies of components as units. For example, if adding/removing an assembly of m parts would make a design better, this would require the manipulation of m elements of the design encoding with a non-generative representation. With a generative representation, abstraction allows for these assemblies to be inserted/deleted through the change of a single symbol, and iteration allows for the addition/deletion of multiple copies of groups of parts through changing the iteration counter. Secondly, reuse of elements of an encoded design allows a generative representation to capture design dependencies by giving it the ability to make coordinated changes in several parts of a design simultaneously. For example, if the diameter of a screw and nut are controlled by the same parameter, then changing the value of the parameter will change the diameter of both screw and nut simultaneously. Or, if all the legs of a table design are a reuse of the same component, then changing the length of that component will change the length of all table-legs simultaneously

```
build-leg()
(
    repeat (5 times) (
        put-brick1
    )
)

build-surface(width) (
    repeat (width times) (
        put-brick2
    )
)

main
(
    put-brick3
    turn-up
    build-leg()
    turn-down
    build-surface(3)
    turn-down
    build-leg()
    put-brick3
)
```

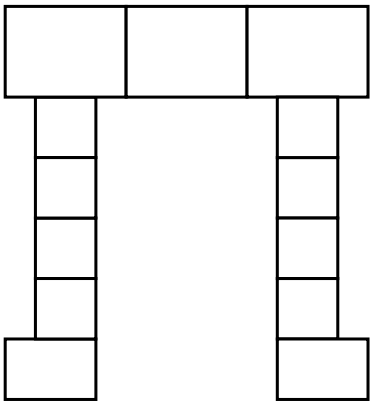
(a) Changing the table leg procedure.

```
build-leg()
(
    repeat (4 times) (
        put-brick1
    )
)

build-surface(width) (
    repeat (width times) (
        put-brick4
    )
)

main
(
    put-brick3
    turn-up
    build-leg()
    turn-down
    build-surface(3)
    turn-down
    build-leg()
    put-brick3
)
```

(b) Changing the table top procedure.



(c) Table created by (a).



(d) Table created by (b).

Figure 1.5: Two changes to the generative representation in figure 1.3.c.

(see the examples in figure 1.4).

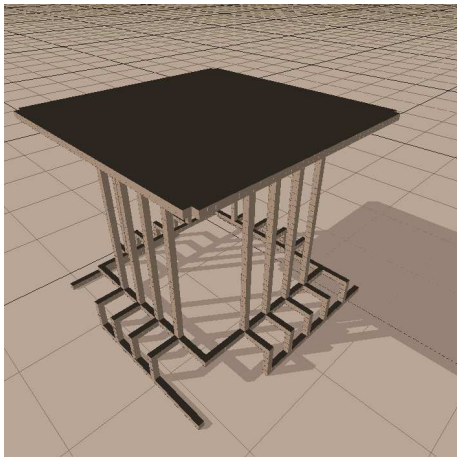
The utility of a generative representation comes in combining it with an evolutionary algorithm. The generative representation provides an open-ended and low-bias method of representing candidate designs and the evolutionary algorithm provides a means of automatically tuning the representation to incorporate the good bias that is learned through the search process. The resulting automated design system is capable of learning dependencies between parts of the design and which assemblies of parts are good for reuse. Thus, for comparable command-sets, evolution with a generative representation should outperform evolution with a non-generative representation.

1.5 Contributions

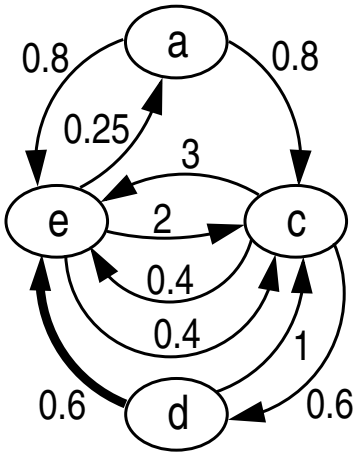
In showing the advantages of generative representations this work makes several contributions to the field of evolutionary design. This section lists the three main contributions.

First, previous work has not identified clear properties of representations. Without clear properties, existing classifications are either ambiguous or do not have boundaries that make meaningful distinctions between different types of representations. Taking computer programming languages as a basis, distinct properties of representations are identified in chapter 2 and, using these properties, representations are classified as either non-generative or generative.

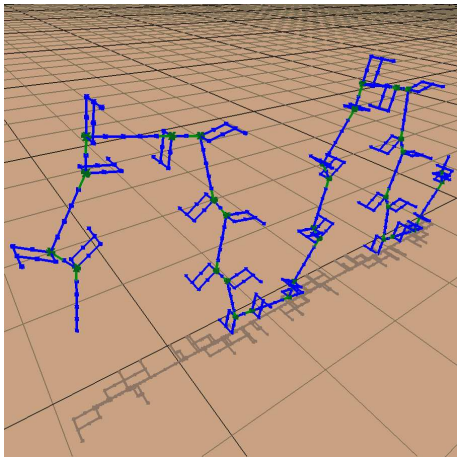
Second, in this work a generic design system called *GENRE*, for *gen*erative *re*presentations, is described. Most evolutionary design systems evolve designs for a single class of structures, consequently there has been no comparisons between representations on multiple classes of designs. The method for producing designs with *GENRE* is based on an explicit, generative representation that generates an assembly procedure for constructing a design. An advantage of this approach is that the evolutionary design system and generative representation can be applied to any problem in which designs are constructed by interpreting the commands of an assembly procedure. This is demonstrated by applying *GENRE* to four



(a)



(b)



(c)

Figure 1.6: Three classes of designs: (a), voxel structures; (b), neural networks; and (c), robots.

different problem domains: voxel structures; neural networks; robots; and neural-network controlled robots. Tables are chosen as a representative of static designs (figure 1.6.a). This design space consists of a 3D array of voxels with dimensions 40×40×40. For the second problem domain, this system is used to evolve neural networks. Evolving neural networks demonstrates that this system is useful not only for creating designs of physical structures, but also of more abstract entities such as software (figure 1.6.b). The third domain is that of creating robots with simple oscillating actuators and the fourth domain consists of combining the design of a robot's morphology along with the design of its controller (figure 1.6.c). In addition, appendix C describes how this system can be used to evolve computer programs to sort lists of numbers.

Finally, it is hypothesized that a generative representation is better than a non-generative representation because the ability to reuse encoded data elements results in more evolvable designs and improves the ability of the evolutionary algorithm to explore large design spaces. This hypothesis is tested by a comparison between the generative and non-generative representations on the four different design problems. Results from this comparison support the claim that evolutionary search with a generative representation will find designs with higher fitness by creating designs with more evolvable encodings and by exploring a wider range of the design space.

1.6 Outline

The remaining chapters of this thesis are organized as follows. In chapter 2, properties of design representation languages are identified and related work in evolutionary design is reviewed. Chapter 3 is a description of *GENRE*, the evolutionary design system used in this thesis. *GENRE* consists of the design constructor and evaluator, the generative representation compiler and the evolutionary algorithm. In addition to describing the different parts of the evolutionary design system, it contains an introduction to Lindenmayer systems, which forms the basis for the generative representation used here. The four different

design domains in which designs are evolved are described in chapter 4. In addition to describing each domain and the command set for constructing designs in them, this chapter gives examples of designs encoded with the generative representation. Chapter 5 presents the results of comparing a non-generative representation to a generative representation on the different problem domains. The results show better performance is achieved with the generative representation and also verify that the generative representation does create, and reuse, components. In chapter 6, these results are used to argue that evolution with a generative representation is capturing intrinsic properties of the design problem, resulting in more evolvable designs and exploration of a larger design space than evolution with a non-generative representation. The ideas and claims of this thesis are summarized in chapter 7, which also presents directions for future work. Finally, the appendices give examples of designs encoded with the generative representation, describe additional work on the domains of two-dimensional robots and sorting programs, and list the source code for compiling the generative representation into an assembly procedure.

Chapter 2

Related Work

Having decided to focus on design representations, it is useful to define some of their properties. Because the mapping from an encoded design to an actual artifact can be considered a computational process, design representations can be thought of as *programming languages*, with encoded designs being *programs* in this language. With this analogy, features of programming languages can be used to understand and classify different approaches to the underlying representations of evolutionary systems. From [2], programming languages have features of:

- **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, recursive calls, but 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/off by the expression of another gene [84]; and an upstream protein can control a downstream protein's activity through a signaling pathway [6].

The meanings of combination, control-flow and abstraction translate almost directly from properties of programming languages to properties of design representations. Combination refers to the ability to create more complex expressions from the basic set of commands in the language. While the subroutines of GLib [8] and genetic programming (GP)[1] allow explicit combinations of expressions, combination is not fully enabled by mere adjacency or proximity in the strings utilized by typical representations in genetic algorithms. Two types of control-flow are conditionals and iterative expressions. Conditionals can be implemented with an `if`-statement, as in GP, or a rule which governs the next state in a cellular automata (CA). Iteration is a looping ability, such as the repeat structure in cellular encoding [51][2], or embedded in the fundamental behavior of CA's. Abstraction is the ability to encapsulate part of the genotype and label it such that it can be used like a procedure, such as with automatically defined functions (ADFs) in GP [82] or automatically defined sub-networks (ADSNs) in cellular encoding. Abstraction can be seen when subfunctions can take parameters, as with ADFs.

Using these properties of programming languages to understand and classify design representations, a fundamental distinction is whether a design representation is *non-generative* or *generative*. With a non-generative representation each representational element of an encoded design can map at most once to an element in a designed artifact. Figure 1.3 shows non-generative representations that map both directly (figure 1.3.a) and indirectly (figure 1.3.b) to a design. Even though the indirect representation of (b) requires a translation process to produce the artifact, it is a non-generative representation because no elements of the encoded design can cause the reuse of an encoding's data elements in constructing the artifact. A **generative representation**, on the other hand, is one with which design encodings are capable of reusing elements of the encoded design in the construction of the

[1] GP is the branch of evolutionary algorithms in which each individual is a computer program [81].
[2] Cellular encoding is reviewed later in this chapter in section 2.3.

artifact. Continuing with the programming language analogy, a generative representation is a kind of language such that elements in the data structure of the encoded design, together with a translation or compilation process, control the flow of execution, allowing for a reuse of elements in the encoded data structure in producing the artifact (figure 1.3.c).

This chapter consists of several section which review the related work in the field of evolutionary design. As the following reviews will show, there has not been a definite trend to more powerful representations. Consequently, this review will group design representations by design substrate and not by class of representation. This categorization also simplifies the review of those design systems which have multiple types of representations – whether for the purpose of comparison or as a result of later work on a given system. The different problem domains into which design systems are categorized are: the evolution of two-dimensional computer images; the creation of shapes; the evolution of neural networks; and the concurrent evolution of robot morphology and controller. The final section is a brief summary of the need for the further investigation of design representations.

2.1 Evolution of Images

The field of evolutionary art has produced a variety of systems which use evolutionary algorithms to produce two-dimensional images. Because of the difficulty in constructing a mathematical function to evaluate the artistic value of an image, the evolution of these systems is usually directed by a user. The user controls evolution by choosing parents for the next generation from a selection of images in the current generation.

One of the first examples of the evolution of images is Dawkins' evolution of two-dimensional pixel shapes, called biomorphs [28]. Biomorphs are encoded in a vector of sixteen real values which encode for the number of branches, segments, scale, and the horizontal and vertical extent of lines. As will be noted later, the images produced by Dawkins' generative representation inspired a number of people to develop systems for the evolution images and shapes. The generative nature of this representation comes from iterative

loops which are controlled by elements in an individual's parameter vector. Not realized in this representation is combination, because there is no method to construct subvectors into atomic units, nor does this representation have conditionals or any form of abstraction.

Another approach to creating computer images is the work of Sims [116] which uses genetic programming [81] to specify the colors of every pixel in an image. In this case, the evolved programs take two parameters, x and y, and return the color index to be assigned to the pixel at those coordinates. The properties which this representation has are combination, which is inherent in the tree-structure of commands, parameterization, and this representation also has an external form of iteration in that the entire program is applied to each pixel. Rooke [106] has extended this approach by including iterative functions, such as the Mandlebrot set, as primitives in the GP language to achieve a more organic look.

Similar to genetic programming, is the computer graphics community's use of Lindenmayer systems (L-systems)[3] as a representation for producing realistic plants [104]. Ochoa used an L-system with a single production rule to produce two-dimensional plants using a fitness function that combines height, weight, branching and ability of leaves to collect sunlight [101]. Jacob used parametric, context-sensitive L-systems to produce plants with leaves at a given range from the base [67], and that maximized the size of a plants shadow and the number of leaves and blossoms [68] [69]. While these systems do not have iteration or conditionals, the different production rules of an L-system are a form of labeled expressions, as well as combination, and parametric L-systems have parameterization and conditionals.

One of the non-programmatic representations is that of the Escher Evolver project [35]. This system uses an evolutionary algorithm to evolve tiled pictures in the style of Escher. Images are represented in a vector of sixty-nine integer values which specify the initial size and shape of a parallelogram, transformations on it, line drawings in the shape, and colors. The resulting object is then tiled to create the final image. This representation has none of the properties of design representations.

The properties of the systems reviewed in this section are summarized in table 2.1. For

[3]Since L-systems are used as the basis for the generative representation in this dissertation they are described in section 3.1.

Table 2.1: Properties of the different representations for evolving images.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
Direct Non-generative					
Escher Evolver [35]	no	no	no	no	no
Explicit Generative					
Biomorphs [28]	no	yes	no	no	no
Jacob [69]	yes	no	yes	yes	yes
Ochoa [101]	yes	no	no	yes	no
Rooke [106]	yes	yes	no	no	yes
Sims [116]	yes	yes	no	no	yes

more details of other evolutionary art projects, see [111] [106] and [135].

2.2 Evolution of Shape

Because there have been a large number of evolutionary design systems for the evolution of shapes, the different design representations are reviewed by type of representation.

2.2.1 Non-Generative Representations for Shape

One obvious method for representing a design is to use a boolean array to specify whether or not there is material at a particular location. An example of this kind of representation is the work of Kane and Schoenaur [72] [73], in which the cross-section of a beam is optimized to maximize its moment of inertia while reducing its weight. A two-dimensional array was used to represent designs with recombination and mutation acting on blocks of these arrays. Another example which directly represented designs with an array is work of Baron in which the two-dimensional cross-section of a beam is evolved to minimize the number of voxels used while falling under the maximum allowable stress [13]. Designs are described by an array of 32×64 bits that is represented by a string of 2048 bits. In both cases, the representation is not able to combine elements of the encoded data structure to create assemblies, nor is it able to manipulate the translation from genotype to phenotype through some form of

control-flow or abstraction.

Schoenauer later compares three different non-generative representations on the evolution of a two-dimensional cantilever design [113] [114]. The first representation is the two-dimensional array representation of [72]. The second representation consists of a set of rectangles that encode for holes in a design. Each rectangle is defined by its height, width and location. The third representation is a set of points that specify a Voronoi diagram. Each point represents a cell and is either the existence of material or empty space. While none of these representations have the ability to combine elements of the genotype or have any form of control-flow or abstraction, the latter two representations (both indirect) more compactly encode designs than the first representation (a direct one).

Two different methods towards more generic design systems are Bentley's system for creating shapes out of cuboids and Roston's grammar based Genetic Design system. Bentley's system uses cuboids – with variable width, height and depth – as a more generic atom for constructing shapes [16] [17]. In addition to specifying the dimensions of a cuboid, the cuboids can also be intersected with a plane of variable orientation allowing for resulting shapes to have surfaces at arbitrary angles. The representation for encoding designs is a tree-structured assembly procedure for attaching cuboids. By using a basic unit of construction, this system was able to evolve ables, portable steps, heat sinks, optical prisms, and streamlined shapes. Roston's Genetic Design (GD) system evolves derivation trees for context sensitive grammars [110]. Interesting in this work is the recognition that by creating a system that uses a general grammar, it can be applied to any design problem in which a grammar can be crafted for creating a design by changing the set of terminals and/or the design constructor. Using his system, Roston evolves frames for the leg positions of an abstract, two-dimensional robot with a simple genetic-programming-style controller and evolves four-bar systems for passing through an ordered set of five points. In both Bentley's and Roston's systems, the use of a tree-structured representation provides an implicit ability to combine elements of the genotype, yet neither representations have the ability to direct the flow of translation through conditionals or iteration, nor do they have any form

of abstraction – although abstraction could be added in a way similar to the automatically defined functions (ADFs) of GP [82] or the automatically defined sub-networks (ADSNs) of cellular encoding (reviewed in section 2.3).

Expanding on functionality is the evolution of buildable structures by Funes [48] [49] [47] using LEGO bricks as the basic component. Similar to the work in evolving trusses, this system tests designs in a simulator to determine if the structure will stay together or fall apart. The representation for designs is a tree-structured encoding that specifies the connection of bricks, starting from a ground position, similar to that of Bentley [16]. As with Bentley's and Roston's systems, the tree-structured representation provides an implicit form of combination but no ability to control the flow of translation nor does it have any form of abstraction.

Another example of a tree-structured representation is the *explicit embryogeny* of Bentley and Kumar [15]. In this representation a tree-structured assembly procedure is used to specify the attachment of basic parts for a two-dimensional, tesselating tile problem. As with Bentley's earlier work [16] and Funes' system [48], this representation has an implicit form of combination, but no forms of control-flow or abstraction.

Another paradigm for the automatic design of shapes is the field Shape Grammars [120]. A shape grammar is a set of rules for transforming a given shape(s) into another shape(s). It differs from the previous systems in that the shapes that are transformed are emergent from the interaction between objects in a design. For example, two overlapping triangles will create at least one more triangle. This last triangle is an emergent result and if the shape grammar has rules for transforming triangles these rules can be applied to this emergent triangle. The typical approach to creating a shape with a shape grammar is to use simulated annealing [75] to iteratively modify a design, encoded with a direct representation, by applying one of the rules in the grammar. Shape grammars have been used to produce a variety of shapes including truss structures [115], coffee machines [3], and MEMS resonators [4]. Difficulties with shape grammars are in developing systems that can automatically recognize all emergent shapes in a design, and then adjusting the representation of elements

in the design to allow for the manipulation of emerged shapes. Since implementations of shape-grammar design systems use a direct representation to store the current state of the design, these representations have no properties of control-flow or abstraction.

2.2.2 Implicit, Generative Representations

Indirect, generative representation consist of a set of design rules which interact to construct a design implicitly, such as with cellular automata and artificial chemistries [31]. These cellular-automata-style rules contain iteration, through the repeated application of construction rules, and conditionals, which are present in the form of determining which rule to apply. As described in the following reviews, these systems can also contain parameters.

One of the earliest methods for generating shapes with computers is the work of Frazer in which the evolutionary design system is a kind of cellular automata for creating objects [42]. His initial work, called *concept seeding*, consisted of producing designs through the application of a series of manually constructed Fortran subroutines which transformed a seed shape by rotating, stretching, growing or shearing it [41]. Initially the GA was used to evolve the seed shape [44], and later it was also applied to the evolution of the transformation rules [42]. Frazer has applied his system to a variety of different shapes such as slabs, columns, and cells in an isospatial grid [43]. This technique, as with all forms of cellular-automata representations, has iteration through the repeated application of the cell-updating rules and uses conditionals to determine which update-rule to apply.

Similar to Frazer's system are the representations of Bentley and Kumar [15] and Bonabeau [20]. In Bentley and Kumar's representation [15], which they call an *implicit embryogeny*, each rule for growing a design consists of a precondition, that specifies when the rule can be applied, and an action to be taken if the precondition is satisfied. Actions can either fill in a neighboring cell or empty the current cell. A design is created by starting from an initial configuration of the grid and then iteratively applying the rules, as with CAs, for a fixed number of iterations. Using this representation, tesselating tile shapes were evolved

in two-dimensional arrays of size 4×4, 8×8 and 16×16. Bonabeau's system [20] is modeled after the construction ability of social insects. Unlike most other systems in which cells are either on or off, cells in this system are able to take on one of multiple state values. The genotype consists of a set of templates for a given state of cells which specify which neighboring cells are to be filled. The fitness function does not evaluate structures for functionality, rather it scores them for containing patterns. Structures were evolved inside a 16 × 16 × 16 grid. These representations have both iteration and conditionals, as with the other cellular-automata-style representations, but no form of combination or abstraction.

More like a traditional cellular automata is that of de Garis' augmented cellular automata [30]. In addition to maintaining a boolean cellular state, each cell also has a set of NEWS variables. These variables store the number of neighbor cells in the ON state in each of the four directions, North, East, West and South. In addition to using cell states to determine which rule to use to update a cell, NEWS values are used as conditionals as part of the automata rules. This use of cell states is not a form of parameterization because these values are state information that is not passed to other cells.

Finally, the property of parameterization is realized by Eggenberger's method of growing three-dimensional shapes from an artificial genome with an artificial chemistry [34]. Shapes are evolved in a 30×30×30 grid, using a measure of an object's symmetry about the x axis as a fitness function. The genome representation for designs is a linear string which consists of regulatory genes, for switching other genes in the genome, and structural genes, which encode for specific chemicals. Each gene consists of six integers, with values in the range of 0-6, that specify its operation within a cell. Shapes evolved with this system have regular structure and symmetry with hundreds of parts. By treating each cell of a shape as an artificial neuron and adding rules for connecting neurons, this system has also been used to evolve neural networks [33]. Parameterization is achieved through the use of artificial substances that affect the switching of genes.

A different method of constructing shapes with a cellular automata is Taura and Nagasaka's Shape Feature Generating Process (SFGP) [123]. In this process, the internal envi-

ronment of a two-dimensional shape is specified, and then growth rules are used to specify the division of dots (metaphors for a cell) on the surface. After development is complete, the final shape is formed by creating an outer surface using the density of dots to determine the distance from the initial shape. Growth rules resemble rules for cellular automata, with the condition based on the density of dots and the action specifying how to divide a dot. Later work has extended this system to three-dimensional shapes [122]. Again, this representation has iteration and conditionals, but no properties of control-flow or abstraction.

2.2.3 Explicit, Generative Representations

Explicit, generative representations are an extension of indirect, non-generative representations by adding reuse through either iteration or abstraction. One example of such an extension is shown in figure 1.3 in which (c) is created by adding iteration, abstraction and parameters to (b).

Expanding on the work of Dawkins (described in section 2.1), Todd and Latham developed Mutator for the evolution of three-dimensional sculptures [128] [126]. With Mutator, structures are defined by an expression in a geometrical construction language that specifies the shape, shape properties, shape transformations, the number of repetitions of a shape and angles between shapes. In the original implementation it is these parameters that are evolved and not the expression. More recently the representational power was increased by adding the ability to evolve the grammar as well as parameters [127]. In this system evolution is guided by a user, who selects which individuals should reproduce from a display of the individuals in the current population. The grammar allows for the hierarchical definition of forms, but sub-forms cannot be labeled for use in other parts of a form expression, consequently this representation has combination and iteration, but not conditionals or abstraction.

Rosenman [107] [109] [108] describes a hierarchical grammar for building two-dimensional, grid-based, floor plans. Unlike other evolutionary design systems, this work uses multiple evolutionary runs to evolve different levels of the design. At the first level, the shape of

rooms are evolved which, once done, become the basic shapes for the next round of evolution, which is the evolution of floor plans. Here, the rules operate on the design itself, and not an assembly procedure. The genotype is a hierarchical assembly plan for building rooms from houses. While individuals do not have the ability to reuse of parts of the grammar through iteration, the ability to build a design from elements evolved in an earlier round of evolution is both a form of combination and labelled procedures.

Broughton, Coates and Jackson [22] [25] [66] use a Lindenmayer system (L-system) as the representation for evolving shapes in an isospatial grid. The L-systems consist of a single production rule, F, and terminals for placing a sphere in each of the twelve sides, for example: $F \rightarrow$ *(F(F POS3(F POS2) F F))*. Using this representation shapes were evolved for a number of different fitness functions consisting of minimizing/maximizing the area projected to a vertical and/or horizontal plane. In addition, a symbiotic coevolutionary experiment was performed in which two L-systems are coevolved, with one L-system evaluated for its ability to enclose space into rooms and the other L-system evaluated for its ability to fill in space. As with the L-system representations for the evolution of images, section 2.1, this representation has combination and labeled expressions.

Another explicit representation, which is a kind of procedural version of Frazer's work [42], is the combination of superquadric modeling primitives [14] with constructive solid geometry[65] [98]. In both systems evolution is guided by a user and was used to produce a variety of shapes such as headless screws [65], and cars and vases [98]. Designs are encoded with a direct network of nodes, with each node specifying a shape-expression. The directed network of nodes allows for shapes to be defined recursively and is a way of adding reuse, through abstraction, to tree-structured representations.

An alternative to constructing a final design from a number of simpler shapes is to define an object by a surface. One system for creating a surface is the work of the Emergent Design group at MIT which uses an L-system for producing a set of points that define a surface [125] [54]. This system uses the standard turtle-graphics command set of L-systems described in [104] to move a turtle in a three-dimensional area. Each stop of the turtle becomes a point

for specifying a surface in this space. Since the turtle turns in angles of 90°, curves are produced by including gravitational attractors/repellers to warp the space.

2.2.4 Summary

Table 2.2: Properties of the different representations for the evolution of shapes.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
Direct Non-generative					
Baron, Tuson and Fisher [13]	no	no	no	no	no
Kane and Schoenauer [72]	no	no	no	no	no
Shape grammar systems	no	no	no	no	no
Indirect Non-generative					
Bentley [16]	yes	no	no	no	no
Bentley and Kumar - *explicit* [15]	yes	no	no	no	no
Funes [47]	yes	no	no	no	no
Genetic Design [110]	yes	no	no	no	no
Schoenauer [114]	no	no	no	no	no
Implicit Generative					
Bentley and Kumar - *implicit* [15]	no	yes	yes	no	no
Bonabeau [20]	no	yes	yes	no	no
de Garis [30]	no	yes	yes	no	no
Eggenberger [34]	no	yes	yes	no	yes
Frazer [42]	no	yes	yes	no	no
SFGP [123]	no	yes	yes	no	no
Explicit Generative					
Broughton, Coates and Jackson [22]	yes	no	no	yes	no
Emergent Design Group [125] [54]	yes	no	no	yes	no
Mutator [126]	yes	yes	no	no	no
Rosenman [109]	yes	no	no	yes	no
Superquadrics [65] [98]	yes	no	no	yes	no

The properties of the different representations for the evolution of shape are summarized in table 2.2, in which they are grouped by which class of design representations they belong to.

2.3 Evolution of Neural Networks

In addition to the evolution of physical designs, EAs have been applied to creating neural networks [137]. EAs have been used to evolve the weights of neural networks, [96] [136], just the network topology [95], and both the topology and the connection weights [10]. This section consists of a review of a representative sampling of the evolution of neural networks in which both the weights and the topology of the networks are evolved.

In [99] [100] a simple developmental model is used for growing neural networks. Networks are created by placing each neuron on a two-dimensional plane and growing connections between them. The representation consists of a sequence of genes, with each gene encoding the parameters for a given neuron. Gene parameters consist of a conditional specifying whether or not the neuron exists, the neuron's properties (such as location), and rules for how to grow links. The growth rules are similar to those of Dawkins [28] and specify the branching angle and segment length for growing a tree from the neuron's location. This representation has both forms of control flow: conditionals, for switching neurons on/off; and iteration in the growth of links from the neurons.

A paradigm for specifying individuals that uses conditionals, but does not have reuse, is the structured GA (sGA) [26]. The individuals of an sGA consist of two strings, A and B, of binary values, in which the first string acts as a switch for turning on/off sections of the second string. The length of the second string is a multiple of the first string – if A is of length S then the length of B is kS – and the value at location A_i specifies whether the k bits starting from B_{ik} are used. In applying this to neural networks, the values in B were used as the weights in a weight matrix and the values in A used to select which connections existed [27]. This is a non-generative representation as there is no reuse of the genotype.

One of the first examples of using a grammar to grow a neural network is the work of Kitano [76]. Kitano's system used an L-system on matrices was used to generate the connectivity matrix of a network. The grammar rules consisted of a non-terminal symbol and a 2×2 matrix of symbols and numbers that it is rewritten with. A consequence of

replacing symbols with a 2×2 matrix is that the the resulting matrices having dimension 2^n. This system was applied to the encoder/decoder problem of sizes 4×4 and 8×8 for which it achieved better convergence with a smaller genotype than did a non-generative representation. This grammar is a generative representation because each grammar rule is a kind of labeled procedure which can be reused.

Another early example of the use of L-systems to generate neural networks is the work of Boers and Kuiper [18] [19]. In this system each terminal symbol in the L-system represents a neuron in the neural network and groupings of symbols inside brackets are used to specify connectivity of the network. Because there is a one-to-one mapping from terminal symbols to particular neurons in the neural network, neural networks with a large number of neurons would require an equally large alphabet of terminals, which limits the ability of this system to scale to large networks.

Similar to the grammatical systems for constructing neural networks is that of cellular encoding in which a network is constructed through the parsing of a tree [51]. Each node of the tree is a network construction command that changes the network by performing an operation on one of the nodes. Reuse of parts of the construction tree comes through iteration. Iteration is done with a recurse node, that indicates that the parsing of the tree should start at the root. This node also contains a counter for controlling the number of times to perform this loop. This system was later extended to include abstraction by using labeled subtrees called automatically defined sub-networks (ADSNs) [50] [52], which are similar to the ADFs of GP [82].

One variation on cellular encoding is edge encoding [89]. With edge encoding, each node in the assembly tree is an operator that acts on an edge (instead of a node) such as by duplicating it, reversing, or splitting in half with a new node in between the two edges. Advantages of edge encoding are that at most one link is created with a construction command so each construction command can specify the weight to attach to that link and, unlike cellular encoding, sub-trees of construction commands will create the same sub-network independent of where in the construction-tree they are located. Other variations

of cellular encoding differ mainly by adding new commands to the network construction language [46] [45].

A synthesis of the growth method of [99] and a tree-structured grammar of [51] is the system of [77]. In this system each node in the tree is a command for creating a neuron, a link, or adjusting the properties of a neuron. Neuron creation commands create a new neuron at a given distance and direction from the parent neuron. Link creation commands create a link between the current neuron and the closest neuron in the specified direction. Evolved networks were applied to the task of controlling the legs of a simulated hexapod to make it walk forward.

Instead of evolving a graph structure, the work of Mautner and Belew used a grammar to iteratively divide a rectangle and interpreted the resulting structure as a network [91] [90]. The grammar, which is like an L-system, consists of non-terminals, which are also rules for splitting a rectangle in two, and terminals, for specifying the final state of a rectangle. Each rectangle in the final structure represents a neuron, with the rectangle's state specifying how it is connected to its neighbors. Evolved networks were used to control a simulated agent in a 500×500 grid-world.

Table 2.3: Properties of the different representations for constructing neural networks.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
Indirect Non-Generative					
sGA [27]	no	no	yes	no	no
Implicit Generative					
Nolfi and Parisi [99]	no	yes	yes	no	no
Explicit Generative					
Boers and Kuiper [18]	yes	no	no	yes	no
cellular encoding [51]	yes	yes	no	yes	no
edge encoding [89]	yes	yes	no	yes	no
Kitano [76]	yes	no	no	yes	no
Kodjabachian and Meyer [77]	yes	no	no	yes	no
Mautner and Belew [90]	yes	no	no	yes	no

Table 2.3 is a summary of the different properties of the representations for the evolution of neural networks.

2.4 Concurrent Evolution of Morphology and Controller

One of the original inspirations for this thesis was the evolution of an artificial creature's morphology and controller. The evolution of artificial creatures has come a long way since Dawkins' evolution of two-dimensional shapes [28]. Serial manipulators have been evolved by evaluating the ability of their end manipulator to achieve a set of configurations [74; 23; 102; 24]; and tree-structured robots have been evolved that met a set of of requirements on static stability, power consumption and geometry [37; 36; 83]. Controllers have been evolved for fixed morphologies: first with stimulus-response rules for animated, articulated creatures [97] [130]; then with neural controllers [53]; and more recently for the dynamic gait of a physical, quadruped robot [56]. 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 [118], as well as competing for the possession of a box [117] – and Ventrella's evolution of stick figures for walking [131]. This has been followed by the evolution of walking creatures by [78] [88] and [21], summarized in [124]. In this section the various systems for the evolution of a virtual robot's morphology and controller are reviewed in chronological order, with a focus on the different representation schemes.

Sims used an embedded, directed graph representation to specify the construction of his creatures [117] [118]. Nodes in the top layer of the graph represent body segments, inside of which 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 [131; 132; 133] 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 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 [80; 78] 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 they 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 [79]. *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 [88], the representation of a creature is the design itself. Both the morphology and 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 [21] 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 [51]. Bongard and Pfeifer report that similarities between 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 gene-rules are applied to the developing creature.

Table 2.4: Properties of the different representations.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
Direct Non-Generative					
Framsticks-*simul* [78]	no	no	no	no	no
GOLEM [88]	no	no	no	no	no
Indirect Non-Generative					
Framsticks-*recur* [78]	yes	no	no	no	no
Implicit Generative					
Bongard and Pfeifer [21]	no	yes	yes	no	no
Explicit Generative					
Framsticks-*devel* [79]	yes	yes	no	no	no
Sims [118]	yes	yes	no	no	no
Ventrella [131]	no	yes	no	no	no

The properties of the different representations used for the evolution of a robot's morphology and controller are summarized in table 2.4.

2.5 Classifications of Design Representations

Classes of design representations and arguments for how to improve scalability have generally used the field of developmental biology for inspiration. While some have argued against existing types of representations they have suggested no alternative beyond pointing to the biological growth of organisms [32] [30]. The work that has looked deeper into design representations has not clearly identified useful properties of representations nor has it supported claims for the advantages of developmental representations. In this section these

investigations into design representations are reviewed.

Angeline [9] classifies representations as translative development functions, generative development functions, and adaptive development functions. With a translative development function the mapping from encoded design to artifact is trivial (one-to-one and independent) or near trivial. This corresponds to a non-generative representation in which the mapping can be either direct or indirect. A generative development function is defined as one with a recursive definition, such as Lindenmayer systems (L-systems) [87], production systems and genetic programming (GP) with automatically defined functions (ADFs) [82]. This definition of generative development functions is similar to generative representations, but the former includes derivation trees for grammars while the latter excludes them because each symbol in the derivation tree is used at most once. Finally, adaptive development functions are defined as those in which the development function can be changed over the course of evolution. The distinction between generative and adaptive development functions is that with a generative development function the change in the development system affects only the individual in which the change was made, whereas with an adaptive development function it can affect all individuals in the population. An example of this is Angeline's work on GLiB [8] in which parts of an individual's genotype can be encapsulated as modules for any other individual in the population to use. Thus a generative representation may not be an adaptive development function because changes in its structure may only affect that individual and an adaptive development function may be non-generative in that each individual could be a derivation tree for a globally defined grammar.

Without comparing different types of design representations, Angeline claims that a representation which uses a development function to map from encoded design to actual design may be more evolvable. By this he means that applying variation to a developmentally encoded individual is more likely to result in a better individual, and large structures can be evolved in less time with developmental representations.

Bentley and Kumar [15] draw inspiration from embryology to classify representations as: no embryogeny, external embryogeny, explicit embryogeny, and implicit embryogeny.

Representations in which there is a one-to-one mapping between elements in the encoded design and elements in the actual design have no embryogeny. This is the same as a direct, non-generative representation and Angeline's translative development function. Embryogenies are those in which there is an indirect mapping between the genotype and attributes of the phenotype are created by multiple elements of the genotype (polygeny). With an external embryogeny, the developmental rules are not changeable by the search algorithms, rather parameters for such a system are evolved. For an explicit embryogeny the rules for creating a design are procedural, such as with genetic programming [81] and Lindenmayer systems [104] but could also be a derivation tree for a grammar in which the nodes of the derivation tree are explicit rules for constructing a design. Thus an explicit embryogeny can be either generative or non-generative. Finally, the development rules of an implicit embryogeny indirectly specify a design, such as with cellular automatas and artificial DNA systems. Implicit embryogenies are generative, because they require the iterative reuse of rules, and correspond to implicit generative representations.

Bentley and Kumar's argument for embryogenies is that they have the benefits of reducing the search space, provide a better enumeration of the search space, allow for more complex artifacts, have repetition and are more adaptable. It is not clear that their embryogenies will lead to scalable design systems with these benefits. Searching in a reduced search space will be faster, but has the possibility of leaving out good solutions and is not a general solution to the scalability problem because it requires the programmer to manually reduce the search space. Similarly, if the mapping between design encoding and artifact is static, then the enumeration of the search space has been chosen by the programmer. Thus while one embryogeny (mapping) may be more adaptable, they admit that another one may be more difficult and it is up to the designer of the embryogeny to pick a good one. Repetition need not be from reuse of the parts of the genotype but may be from a non-generative representation in which multiple parts of the genotype use the same translation rule, such as with the grammars in Roston's system [110]. Their experiments with the different embryogenies on a two-dimensional tiling problem of various sizes compare results with respect to

fitness and encoding size. In their comparison the implicit embryogeny scaled best, followed (in order) by the non-embryogeny, the explicit embryogeny and the external embryogeny; showing that an embryogeny may not be more scalable than a non-embryogeny. This does not fully support their claim that embryogenies are better than non-embryogenies. Using a generative/non-generative classification the results show that reuse is a good property because the generative representation (here the implicit embryogeny) finished ahead of the three non-generative representations.

Komosinksi and Rotaru-Varga [79] compare three different representations on three different problems within the class of articulated creatures. Although they use different terms, their three representations match the style of representations in figure 1.3: *simul*, a direct low-level encoding, matches the direct representation in figure 1.3.a; *recur*, which they call a direct recurrent encoding is an indirect, non-generative representation matching that in figure 1.3.b; and *devel*, an indirect developmental encoding, is of the style of the representation in figure 1.3.c. The characteristics they identify for these representations (rated in terms of none/low/variable/high) are: genotype complexity; interpretation complexity; body constraints; brain constraints; modularity; compression; and redundancy. Of these, only modularity and redundancy are comparable with the properties listed at the start of this chapter. Comparing these three representations on the separate tasks of optimizing height and velocity, it was found that *recur* and *devel* were better then *simul*, but found little difference between *recur* and *simul*. Since evolved designs tended to have few parts with all three representations it is likely that the generative representation afforded little advantage because either the design space was not conducive to large designs (neither *recur* or *simul* were able to generate creatures with closed loops) or the problems did not require complex solutions. The authors concluded that representations should be high-level and structured.

In summary, the classification systems and representations of previous work have both similarities and differences to that of generative/non-generative representations. The classification systems of Angeline [9] and Bentley and Kumar [15] recognize that an indirect

relationship between the encoded design and resulting artifact is necessary for design representations to scale but neither recognize reuse of data elements of the encoded design as an essential property for scalability. A representation that separates their classifications from that of generative representations is a simple derivation tree for a grammar. This mapping qualifies as a developmental function and an embryogeny – it may even result in repetition in the artifact through the reuse of grammar rules – but because it does not allow for parts of the derivation tree itself to be reused it is not a generative representation. While the properties which Komosinski and Rotaru-Varga [79] give for design representations do not match those given in this dissertation, the three representations they use are examples of a direct, non-generative representation, an indirect, non-generative representation and an indirect, generative representation.

2.6 Summary

In this chapter related work in evolutionary design and design representations was reviewed. Comparing the representations from different classes shows that each class tends to have a different set of properties. While the representations of one of the implicit, generative representations is parameters, not present in any of these systems is the ability to combine construction rules or procedure labels. Compared to implicit representations, the explicit, generative representations have three advantages from using a procedural approach for constructing designs. First, an explicit, procedural approach allows for the property of combination: assemblies of parts can be encapsulated into a unit which can then be readily transfered from one individual to another. Second, an explicit representation allows for a modularization of the representation such that different parts of the artifact encoding create different parts of the artifact. This modularization enables changes to be made to one part of an artifact without affecting other parts, unlike with an implicit representation in which this ability is inhibited by the interaction between construction rules. The third advantage with a procedural approach is that the separation between the encoding and the construc-

tion of a design results in a generic design system: different classes of design can be evolved by merely changing the set of construction commands and the design builder, [110].

While a wide variety of representations have been used for many different problems, there is little guidance for how to construct a representation that will scale to complex designs. With the exceptions of Roston's Genetic Design system [110], and Soddu's Argenia [119], design systems have been used for creating only single class of designs. Consequently, for those design systems that do produce interesting designs, part of the reason may be from a tight coupling of the design system to the design space and the representation may not be transferable to a different class of designs. Previous examination of design representations suggest only that a representation be indirect [9] [15] and high-level [79]. Thus the goals of this dissertation – the identification of different properties of design representations and showing the importance of reuse – address the need for further investigation into design representations.

Chapter 3

Methods

The evolutionary design system used to compare a generative representation to a non-generative representation is *GENRE*. This system consists of the design constructor and evaluator, the compiler for the generative representation, and the evolutionary algorithm. Each design is specified by a sequence of construction commands for building the artifact, called an *assembly procedure*. For the non-generative representation, an individual's genotype is an assembly procedure. With the generative representation, each individual consists of a program which is then compiled into an assembly procedure. This programming language for encoding assembly procedures is based on Lindenmayer systems (L-systems) [104], which are a type of grammar for producing sequences of characters (strings). By using an indirect, non-generative representation and an explicit, generative representation, these two representations can be applied to different design substrates by changing only the set of construction commands and the design constructor.

The first two sections of this chapter describe the two representations used by *GENRE* and the evolutionary algorithm used to evolve designs. In describing the representations used in this thesis, section 3.1 begins with an introduction to L-systems and expands on this description to describe how L-systems are used as the generative representation. This is followed by a description of the non-generative representation. The second section describes the details of the particular evolutionary algorithm used for evolving designs. Since the

canonical evolutionary algorithm was already described in section 1.1, this section goes into detail on the parts of the EA that are specific to this representation; namely how the population is initialized and how mutation and recombination operate on the representations.

3.1 Generative Representations

In this thesis, each design is encoded as a Lindenmayer system (L-system) [86]. L-systems are a grammatical rewriting system introduced to model the biological development of multicellular organisms, with rules applied in parallel to all characters in the string just as cell divisions happen in parallel in multicellular organisms. Complex objects are created by successively replacing parts of a simple object by using the set of rewriting rules. This section begins with an introduction to basic L-systems, then describes parametric L-systems, the class of L-systems used in this thesis, and concludes with a description of how L-systems are used as a generative representation.

3.1.1 Basic L-systems

L-systems are a class of rewriting systems. Rewriting consists of going through a sequence of symbols and replacing each symbol with another symbol, or subsequence of symbols, and a rewriting system consists of a set of rules for specifying how symbols are rewritten. By iteratively applying the set of rewrite rules from a starting symbol, a complex string is created from a succession of simpler ones. For example, the L-system,

$$a: \quad \rightarrow a\ b$$
$$b: \quad \rightarrow b\ a$$

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

$$a$$

$$ab$$

$$abba$$

$$abbabaab$$

3.1.2 Turtle Interpretation and Bracketed L-systems

One application of L-systems is as a method for generating natural-looking plants [104]. This creates drawings by interpreting the symbols of the string produced by an L-system as commands for a LOGO-style turtle [1].

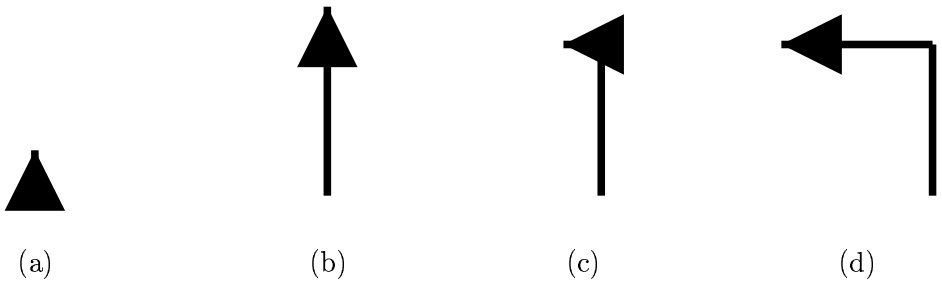


Figure 3.1: Drawing with a turtle.

An example of drawing with a turtle is shown by the sequence of images in figure 3.1. The language used by this turtle is: `f`, moves the turtle forward; `l`, turns the turtle 90° to the left (counter-clockwise); and `r`, turns the turtle 90° to the right (clockwise). Executing the command sequence `flf` produces the sequence: the starting state, figure 3.1.a; executing `f`, figure 3.1.b; executing `l`, figure 3.1.c; and executing `f`, figure 3.1.d. Thus the command sequence `flf` is an assembly procedure for creating the line in figure 3.1.d.

A shortcoming of this drawing language is that the turtle can only draw a single, unbroken line. To add a branching ability to turtle drawing, brackets, '[' and ']', are used to store/retrieve the turtle's state – which consists of its location and heading – to/from a stack. Thus interpreting the sequence `f[lf][rf]f` produces the sequence of drawings in

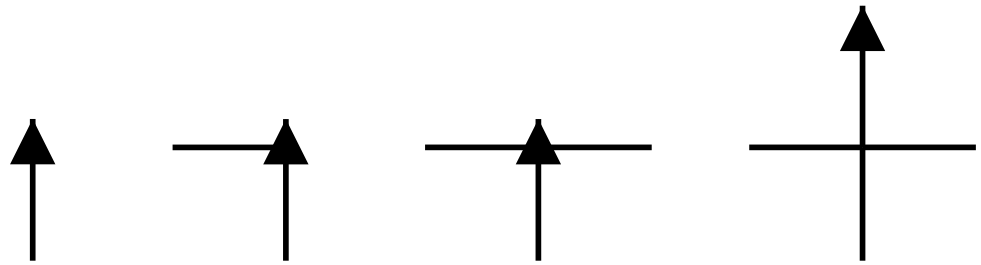


Figure 3.2: Drawing with a turtle, using brackets.

figure 3.2. The command `f` moves the turtle forward, figure 3.2.a. In executing `[lf]` the current state of the turtle is pushed onto a stack by the opening bracket `[`, the commands `lf` turn the turtle left and move it forward, then the stored turtle-state is popped off the stack with the command `]`, figure 3.2.b. Similarly, the commands `[rf]` push the turtle's state to the stack, turn it right and move it forward, then restore the saved state, figure 3.2.c. The final command `f` moves the turtle forward, figure 3.2.d.

Table 3.1: Design language for voxel structures.

Command	Description
f	Move the turtle forward.
l	Rotate the turtle's heading $\delta°$ to the left.
r	Rotate the turtle's heading $\delta°$ to the right.
[ ]	Push/pop state to stack.

The commands `f`, `l`, `r`, and `[]` are more generally defined in table 3.1. With these commands the amount of rotation caused by the commands `l` and `r` acan be controlled by changing the constant δ. An example L-system that uses this command language is,

$$A : \ \rightarrow f\ [\ l\ A\ ]\ [\ r\ A\ ]\ f\ A$$

This L-system produces the following sequence of strings,

1. *A*
2. *f[lA][rA]fA*
3. *f[lf[lA][rA]fA][rf[lA][rA]fA]ff[lA][rA]fA*
4. *f[lf[lf[lA][rA]fA][rf[lA][rA]fA]ff[lA][rA]fA][rf[lf[lA][rA]fA][rf[lA][rA]fA]*
   ff[lA][rA]fA]ff[lf[lA][rA]fA][rf[lA][rA]fA]ff[lA][rA]fA

By ignoring the symbol `A`, these strings can be used to create drawings. With δ set to 30°, these drawings are shown in figures 3.3.a through d. This L-system can be used to produce a variety of tree-like images by coloring the lines brown and green and using different for δ. The images in figure 3.3.e and d show tree-like images for two different values of δ in which lines which have children branching from them are colored brown and lines which have no children are colored green.

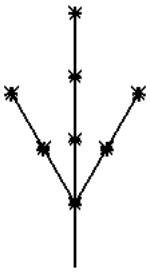
3.1.3 Parametric L-systems

Another extension of basic L-systems is the class of parametric L-systems [87] (PL-systems). This class differs from basic L-systems in that the production rules of PL-systems have parameters, there can be algebraic expressions applied to parameter values and parameter values can also be used in determining which production rule to apply. A production rule consists of three components: the predecessor, the condition and the successor. For example, a production with predecessor A(n_0, n_1), condition $n_1 > 5$ and successor B(n_1+1) c D(n_1+0.5, n_0-2) is written as:

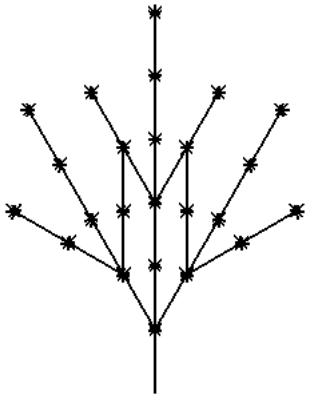
$$A(n_0, n_1) : n_1 > 5 \rightarrow B(n_1+1)cD(n_1+0.5, n_0-2)$$

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

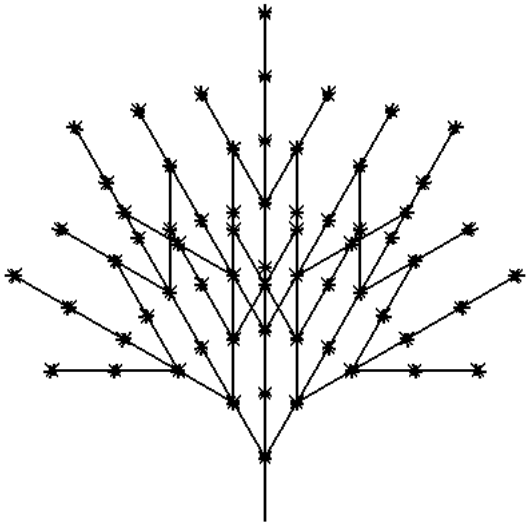
(a) $\delta = 30°$, iteration 2.



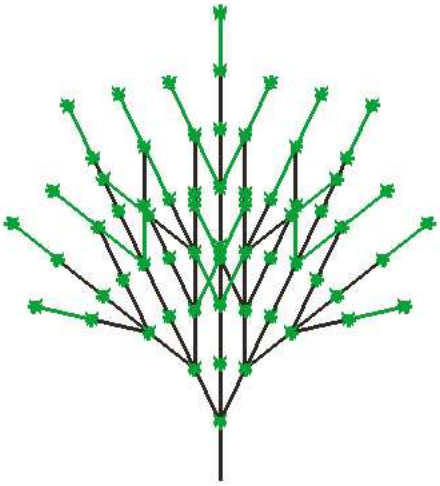
(b) $\delta = 30°$, iteration 3.



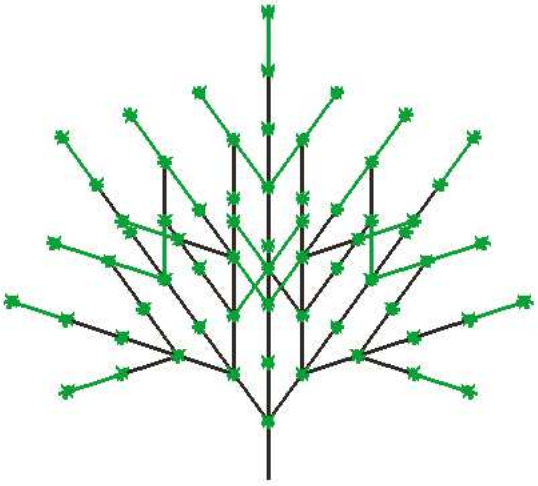
(c) $\delta = 30°$, iteration 4.



(d) $\delta = 30°$, iteration 5.



(e) $\delta = 25.7°$, iteration 5.



(f) $\delta = 36°$, iteration 5.

Figure 3.3: Trees constructed from the L-system in section 3.1.2.

For example, the PL-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 $a(4)$, produces the following sequence of strings,

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

Previously EAs have been combined with L-systems to evolve neural networks [76] [19], plants [68] [101] and architectural structures [25]. For the most part, this past work has used *non-parametric* L-systems whereas here *parametric* L-systems are used. An advantage of a parametric L-system over a non-parametric L-system is that a given PL-system can produce a family of strings, with the specific string determined by the starting parameter(s). For example, the parameter to a production rule can be used as the argument to 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 to a network construction command can specify the weight of a newly created link in the network or the parameter to a rod-creation command can specify the rod's length.

3.1.4 L-systems as a Generative Representation

Of the properties of design representations listed in section 2.5, P0L-systems have conditionals, abstraction and parameters. The generative representation of this thesis includes iteration through a looping ability that is new to L-systems. Looping is performed by the replication of symbols within enclosed parenthesis, so that $\{\ block\ \}(n)$ repeats the enclosed block of symbols n times. Block replication is similar to *for-next* loops in computer programs and is almost identical to the multiple rewriting of the recurrent symbol of cellular encoding [52] and the recursive-limit parameter in graph encoding [117]. For example $\{abc\}(3)$ translates to, *abcabcabc*. In this implementation the unraveling of block-replication loops is left until the final iteration. For example the generative representation,

$$P0(n_0): \quad n_0 > 1.0 \rightarrow [\ P1(n_0 * 1.5)\ ]\ a(1)\ b(3)\ c(1)\ P0(n_0 - 1)$$

$$P1(n_0): \quad n_0 > 1.0 \rightarrow \{\ [\ b(n_0)\ ]\ d(1)\ \}(4)$$

when started with $P0(4)$ produces the following sequence of strings,

1. *P0(4)*
2. *[P1(6)] a(1) b(3) c(1) P0(3)*
3. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [P1(4.5)] a(1) b(3) c(1) P0(2)*
4. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [P1(3)] a(1) b(3) c(1) P0(1)*
5. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [{ [b(3)] d(1) }(4)] a(1) b(3) c(1)*
6. *[[b(6)] d(1) [b(6)] d(1) [b(6)] d(1) [b(6)] d(1)] a(1) b(3) c(1) [[b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1)] a(1) b(3) c(1) [[b(3)] d(1) [b(3)] d(1) [b(3)] d(1) [b(3)] d(1)] a(1) b(3) c(1) b(3)*

For implementation reasons constraints are added to the P0L-system. The condition is restricted to be comparisons as to whether a production parameter is greater than a

constant value. Parameters to design commands are either a constant value or a production parameter. Parameters to productions are equations of the form: *[production parameter | value] [+ | - | × | \ ] [production parameter | value]*.

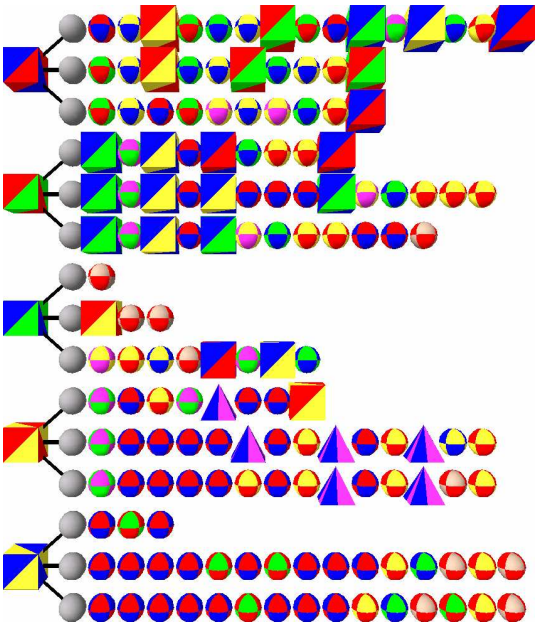
In addition to storing the grammar of an L-system, each individual in the EA also stores the starting condition, or seed, and the number of rewriting iterations. The first production rule, *P0*, is always used as the starting symbol, consequently the starting condition consists of the initial arguments to this rule. After performing the specified number of rewriting iterations, the resulting string of symbols is stripped of all production rule symbols so that it consists of only construction commands, resulting in an assembly procedure for building a design.

To create designs with these L-systems, the non-production symbols are interpreted as construction commands in a design construction language. After undergoing a given number of iterations of rewrites, the final string produced is passed on to a design constructor to build the design. Using the key: (a = up; b = forward; c = down; and d = left); this L-system is the same as that of the example in section 4.1.1, which shows how this L-system produces a voxel-based design. Thus the resulting system is a kind of programming language with loops, function-calls and a user-specified command-set.

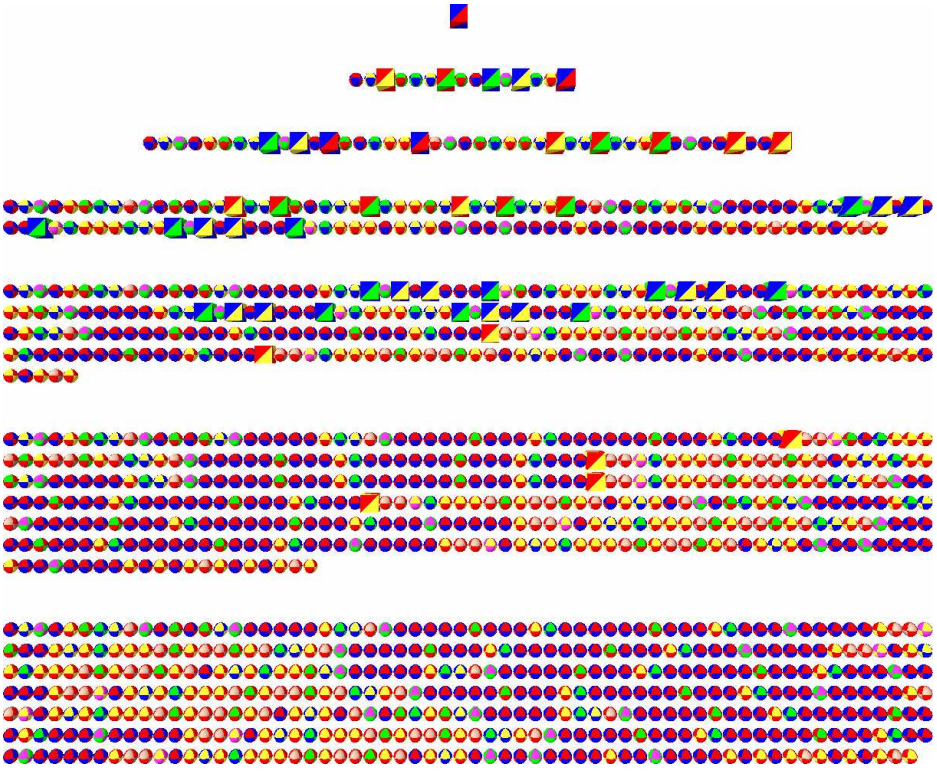
Figure 3.4.a shows a graphical representation of the rules for our generative representation. In these images cubes represent procedure calls, grey spheres represent conditionals, pyramids represent the repeat operator and spheres represent construction commands (such as those listed in chapter 4). Figure 3.4.b 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.

3.1.5 L-systems as a Non-Generative Representation

The non-generative representation is implemented as an L-system with one production rule, no arguments, one condition-successor pair whose condition always succeeds, and without



(a)



(b)

Figure 3.4: Graphical rendition of the generative representation, (a), along with the sequence of strings produced, (b).

the repeat operator (iteration) or the ability to call production rules (abstraction). Removing abstraction and control-flow from an L-system reduces it to a degenerate case in which it is a single string of terminal symbols. By implementing the non-generative representation as a degenerate case of the generative representation, the evolutionary design system is able to use the same variation operators on both representations so that the only difference between the two representations is the ability to reuse elements of the genotype.

3.2 Evolutionary Algorithm

The evolutionary algorithm used to evolve designs is the canonical generational EA with specialized variation operators. A non-generative representation and a generative representation are used to encode designs, both of which are implemented as L-systems. The L-system for the non-generative representation is different from the generative representation in that it does not have conditionals or the ability to re-use part of the encoding through iteration or abstraction. The non-generative representation is implemented as a single production rule, with no arguments, containing 1 condition-successor pair (whose condition always succeeds), and without the repeat operator or the ability to call production rules. Implementing both non-generative and generative representations in the same way allows the same initialization procedure and variation operators to be used on both representations. The initial population of L-systems is created by making random production rules. After individuals are evaluated, their probability of being selected as parents is calculated using exponential scaling [94] and then parents are selected using stochastic remainder selection [12] with an elitism of two. New individuals are created through applying mutation or recombination (chosen with equal probability) to individuals selected as parents. This process of evaluation, selection, and reproduction is then repeated for a fixed number of generations. In addition, data is kept for each L-system as to 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 which were used. It also allows historical-based constraints on the mutation of conditional values. Since variations sometimes create an invalid robot (with too many/few rods, or 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). As initialization, mutation and recombination are dependent on the representation these are now described in greater detail.

3.2.1 Initialization

An initial L-system is created from a blank template of a fixed number of production rules, each with a fixed number of condition-successor pairs. Conditions are created by randomly picking a parameter and a constant value to compare against. Successors are created by stringing together sequences of randomly generated blocks of one to three characters, which may be enclosed by push and pop symbols, '[' and ']', or block replication symbols, '{' and '}'. Examples of initial blocks of characters are: *a(2.0) b(3.0) c(4.0)*, *{ P2(*n_1*+2.0,*n_1*/3.0) d(3.0) }(2)*, and *[a(*n_0*)]*.

After an L-system is created, it is evaluated. L-systems that score below a preset threshold are discarded and a new one is randomly created in its place. This way the initial population consists of a variety of different solutions, each of which is a design whose fitness is above some minimal value.

3.2.2 Mutation

Mutation creates a new individual by copying the parent individual and making a small change to it. To mutate the L-system, a production rule is selected at random from one of the used production rules. Possible changes that can occur are: replacing one command with another command, generated at random; perturbing the parameter of a command by adding/subtracting a small value to it; changing the parameter equation in a production rule; adding/deleting a block of commands in a successor; changing the condition equation; or

encapsulating a sequence of commands and placing them in a previously unused production rule. In addition, an individual can be mutated by perturbing one of the seed parameters used to start the L-system or by recombining it with itself.

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

$$\begin{aligned} P0(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \end{aligned}$$

some of the possible mutations are,

Mutate an argument:

$$\begin{aligned} P0(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(\mathbf{n_1 - 2.0}, n_0/2.0)\ ] \end{aligned}$$

Delete random character(s):

$$\begin{aligned} P0(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ \}(n_1)\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \end{aligned}$$

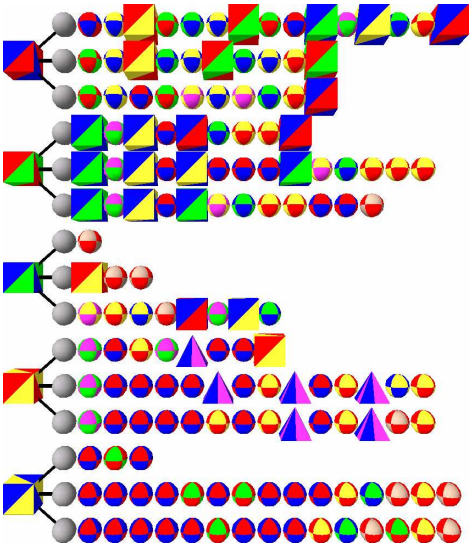
Insert a random block of 1-3 character(s):

$$\begin{aligned} P0(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ \mathbf{e(4.0)}\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \end{aligned}$$

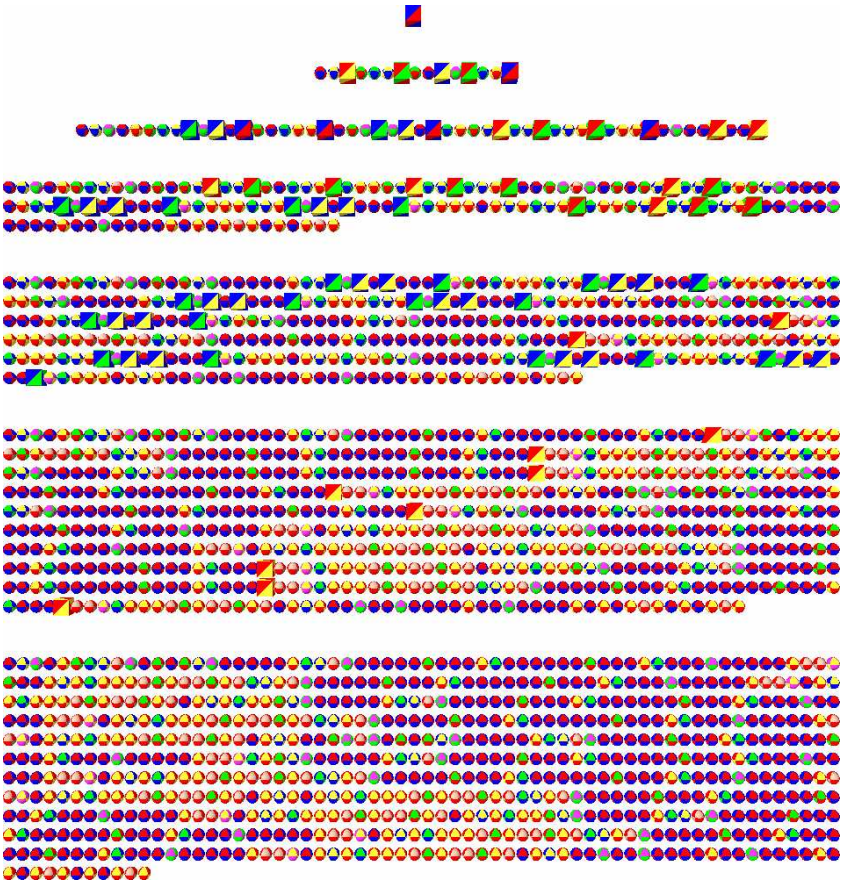
Encapsulate a sequence of character(s):

$$\begin{aligned} P0(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \mathbf{P2(n_0, n_1)}\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \\ P2(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1) \\ & n_0 > 2.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1) \end{aligned}$$

To illustrate how small changes in the genotype can produce large changes in the pheno-



(a)



(b)

Figure 3.5: A single mutation to the individual in figure 3.4 produces the L-system in (a), which produces the sequence of strings in (b).

type, an example of swapping the order of two symbols is shown in figure 3.5. The L-system in figure 3.5.a has the order of two symbols in the second successor of the second production symbol, resulting in large changes in the phenotype, as can be seen by comparing the strings produced: figure 3.4.b contains the original sequence of strings and the results after the change is shown in figure 3.5.b.

3.2.3 Recombination

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

For example if parent 1 has the following rule,

$$\begin{aligned} P3(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ c(3.0) \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \end{aligned}$$

and parent 2 has the following rule,

$$\begin{aligned} P3(n_0, n_1): \quad & n_1 > 3.0 \rightarrow b(3.0)\ a(2.0)\ c(1.0) \\ & n_0 > 1.0 \rightarrow P1(n_1 - 1.0, n_1 - 2.0) \end{aligned}$$

Then some of the possible results of a recombination on successor P3 are:

Replace an entire condition-successor pair:

$$\begin{aligned} P3(n_0, n_1): \quad & \mathbf{n_1 > 3.0 \rightarrow b(3.0)\ a(2.0)\ c(1.0)} \\ & n_0 > 2.0 \rightarrow d(4.0)\ [\ P1(n_1 - 1.0, n_0/2.0)\ ] \end{aligned}$$

Replace just a successor:

$$\begin{aligned} P3(n_0, n_1): \quad & n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ c(3.0) \\ & n_0 > 2.0 \rightarrow \mathbf{P1(n_1 - 1.0, n_1 - 2.0)} \end{aligned}$$

Replace one block with another:

$$P3(n_0, n_1): \quad n_0 > 5.0 \rightarrow \{\ a(1.0)\ b(2.0)\ \}(n_1)\ c(3.0)$$
$$n_0 > 2.0 \rightarrow d(4.0)\ [\ \mathbf{b(3.0)}\ \mathbf{a(2.0)}\ ]$$

3.3 Summary

Table 3.2: Properties of *GENRE*'s non-generative and generative representations.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
GENRE: indirect non-generative	yes	no	no	no	no
GENRE: explicit generative	yes	yes	yes	yes	yes

In this chapter the non-generative and the generative representation were described, as well as the evolutionary algorithm that was used to evolve designs with them. The properties of these representations are listed in table 3.2. Unlike the generative representations reviewed in chapter 2, *GENRE*'s generative representation 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.

Chapter 4

Design Domains

The generative representation system described in chapter 3 is generic and can be applied to different design domains by using a different command set and/or design constructor. This chapter describes four different design domains that are used in chapter 5 to compare the non-generative and generative representations. The first class of designs consists of static structures built from voxels. The second class of designs is that of neural networks. The third class of designs consists of robots, constructed from Tinker-Toy^{TM}-like parts, with actuated joints that oscillate through a fixed range. Combining the second and third classes results in neural-network controlled robots, which is the fourth class of designs. In this chapter the languages used for these four classes of designs are described.

4.1 Voxel Structures

The first class of designs consists of three-dimensional static objects constructed of voxels (cubes). Commands in the command set are instructions to a LOGO-style turtle, and are listed in table 4.1. For the construction of three-dimensional shapes, a three-dimensional matrix is used to store the absence/presence of material at a particular location, similar to the methods of [72] and [13]. This matrix starts out empty and voxels are filled when the turtle enters them. The commands '[' and ']' push and pop the current state – consisting

Table 4.1: Design language for voxel structures.

Command	Description
[ ]	Push/pop state to stack.
forward(n)	Move in the turtle's positive X direction n units.
back(n)	Move in the turtle's negative X direction n units.
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.
counter-clockwise(n)	Rotate heading $n \times -90°$ about the turtle's X axis.

of the current position and orientation – to and from a stack. Forward moves the turtle forward in the current direction and back moves the turtle backwards, both place a block in the space if none exists. Turn left/right/up/down/clockwise/counter-clockwise rotate the turtle's heading about the appropriate axis in units of 90°.

The images in figure 4.1 show intermediate stages in the construction of an object. Initially there is a single cube in the design space, figure 4.1.a. After executing the commands `forward(2)`, two cubes are added to the first, figure 4.1.b. The image in figure 4.1.c shows the design after executing `right(1) forward(1)`, which turns the orientation of the turtle 90° to the right and then adds a cube after moving forward one space. The final image, after executing `up(1) forward(3)`, is shown in figure 4.1.d.

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

4.1.1 Generative Representation Example for Voxel Structures

The following is an example encoding of a design using the generative representation and the construction language of table 4.1 for building designs with voxels. It consists of two

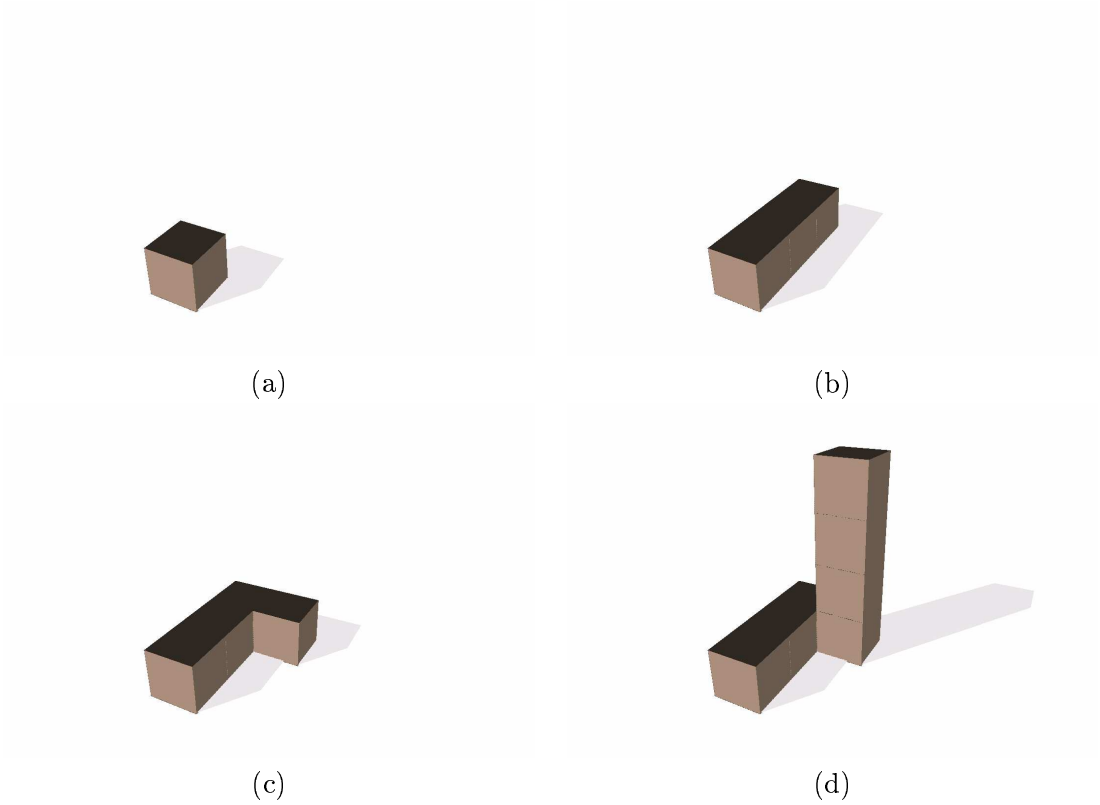


Figure 4.1: Building an object.

productions with each production containing one condition-successor pair:

$$P0(n0): \quad n0 > 1.0 \rightarrow [\ P1(n0 * 1.5)\ ]\ up(1)\ forward(3)$$
$$down(1)\ P0(n0 - 1)$$

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

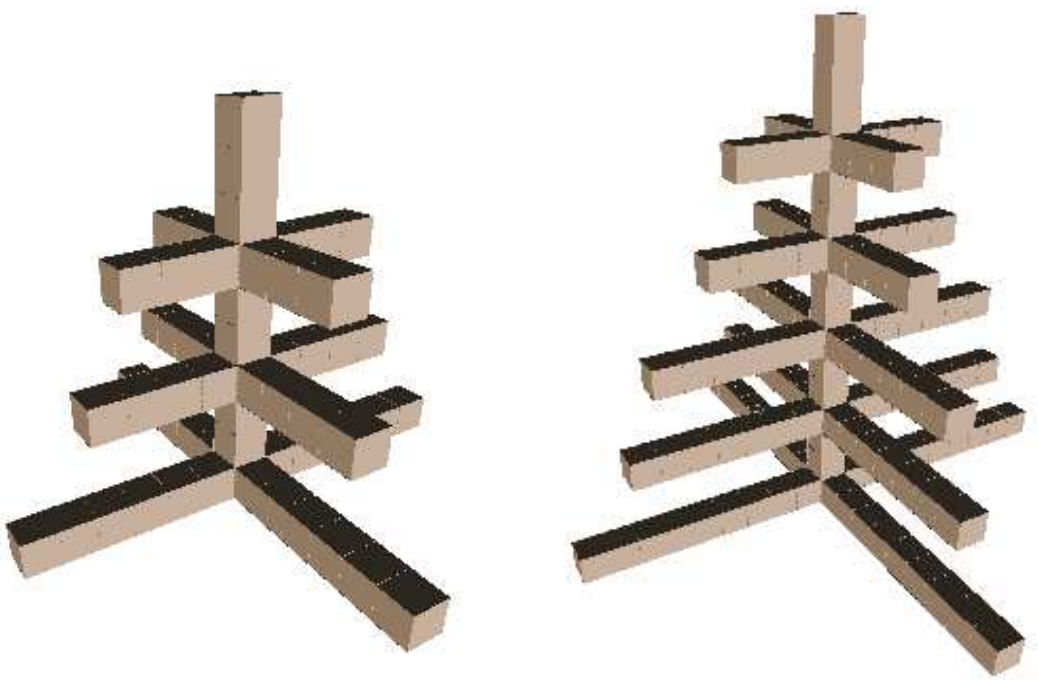
Interpreting the build commands as controls for a turtle in a three-dimensional voxel world this L-system creates the trees in figure 4.2. Starting this design encoding with P0(4), produces the following sequence of strings,

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

Executing this string with the design constructor produces the structure shown in figure 4.2.a. Trees of arbitrary size can be created by starting the production system with a different argument – the tree in figure 4.2.b is created from this system by starting it with P0(6).

4.2 Neural Networks

The method for constructing the neural controllers for the artificial creatures is based on that of cellular encoding [51]. The main difference is that build commands operate on the links connecting the nodes, as with edge encoding [89], 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 sub-sequences of build commands will construct the same sub-network independent of their location in the



(a) (b)

Figure 4.2: Two example structures.

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 [86] 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 index of the links into these neurons – to and from the stack. This stack of edges allows branching to occur in the 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. The commands for this language are listed in table 4.2, for which the current link connects from neuron A to neuron B.[1]

[1]The description here of split() differs from [60; 62] as to which link takes on which weight value – the definition in table 4.2 correctly matches the implementation.

Command	Description
[ ]	Push/pop state to stack.
add-input(n)	Creates an input neuron with a link from it to neuron B with weight n.
add-output(n)	Creates an output neuron with a link from B to it with weight n.
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.
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 C to B and creates a new link connecting from neuron A to neuron C with weight n.

Table 4.2: Design language for neural networks.

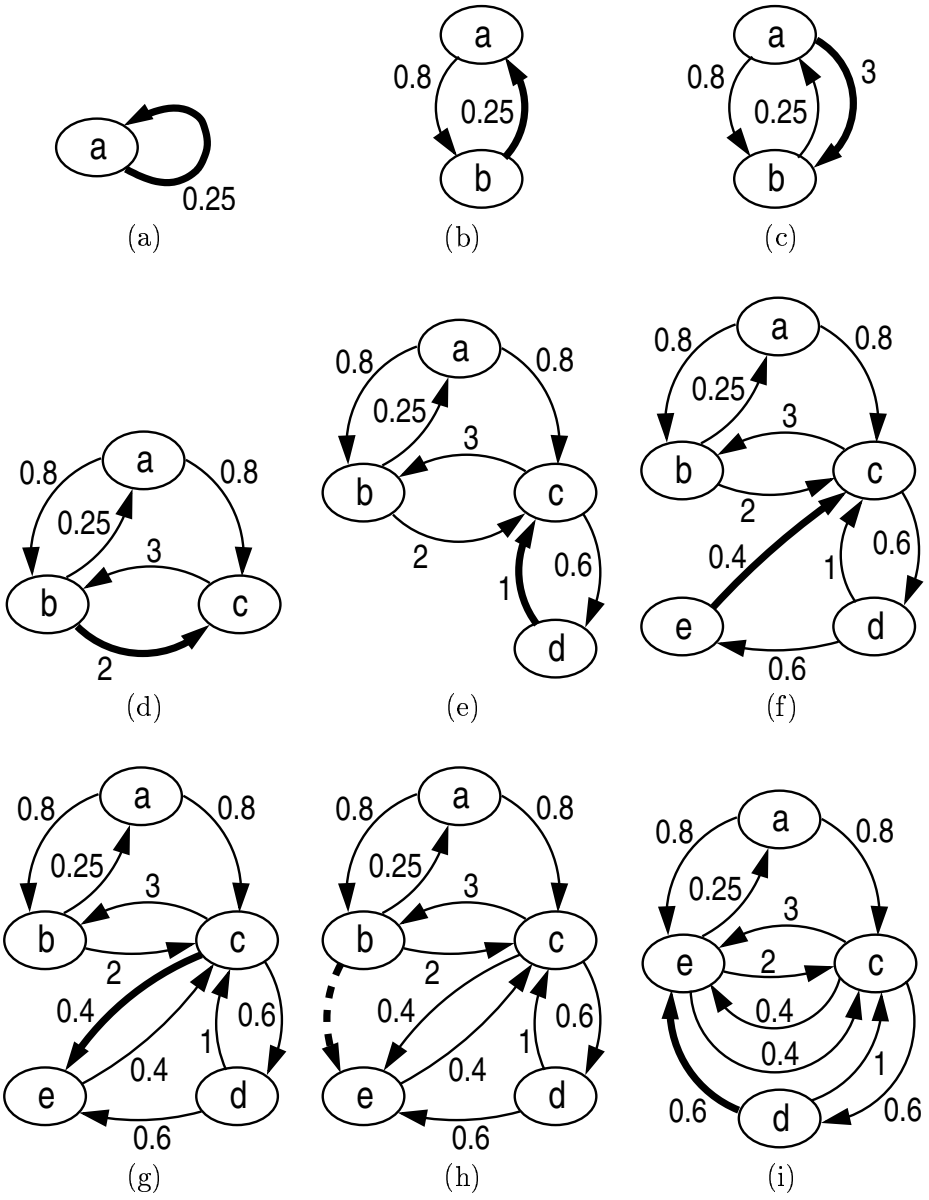


Figure 4.3: Construction of a neural network.

An example of the construction of a network using this system is shown in figure 4.3, 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 4.3.a. 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 4.3.b. Executing *duplicate(3)*, creates a second link from the second neuron to the first, which is then reversed in executing *reverse*, figure 4.3.c. The execution of *split(0.8) duplicate(2) reverse*, creates a third neuron, figure 4.3.d. 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 4.3.e. 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 4.3.f. Another link is created from e to c with *duplicate(0.4)*, which is then reversed, *reverse*, figure 4.3.g. *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 4.3.h. These two neurons are then joined together by the *merge(1)* command, and the final network is shown in figure 4.3.i. Once interpreting the assembly procedure has finished, networks are simplified by combining the weight values of links with identical source and destination neurons. Simplifying the network in figure 4.3.i results in the weight matrix shown in table 4.3.

Table 4.3: Weight matrix.

Input	Output			
	a	c	d	e
a	0.0	0.8	0.0	0.8
c	0.0	0.0	0.6	3.4
d	0.0	1.0	0.0	0.6
e	0.25	2.4	0.0	0.0

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

Table 4.4: Sequence of activation values.

Iteration	a	c	d	e
0	1.0	0.0	0.0	0.0
1	0.990000	0.659541	0.376285	0.997058
2	0.770735	0.997711	0.536072	0.999654
3	0.780087	0.998382	0.536358	0.999660
4	0.790085	0.998408	0.536370	0.999666
5	0.800084	0.998433	0.536380	0.999671

outputs clipped to the range ± 1. The different transfer functions are: sigmoid, using $tanh(sum\ of\ inputs)$; linear; and an oscillator. Oscillator units maintain a state which 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 in [78] [88] it is a less biased model than that of [131], in which all actuators are driven by oscillators, or [118] which used a variety of transfer functions and oscillating neurons. The initial activation value for neurons with the sigmoid and linear transfer functions is 0.0 and the initial activation value for oscillator units is 1.0. An example of the sequence of activation values for a network is shown in table 4.4, which are the sequence of activation values for the network from the above example.

4.3 Oscillator Controlled Robots

The third class of designs consists of a Tinker-Toy^{TM}-like world in which robots are built from rods of regular length and both fixed and actuated joints, figure 4.4. The turtle command set for three-dimensional genobots is an extension to the 2D command set of the previous section. *Push* and *pop* operators, '[' and ']', are used to store and retrieve the current state – consisting of the current location, orientation, and relative phase offset – to and from the stack. *Turn left/right/up/down/clockwise/counter-clockwise(n)* rotate the

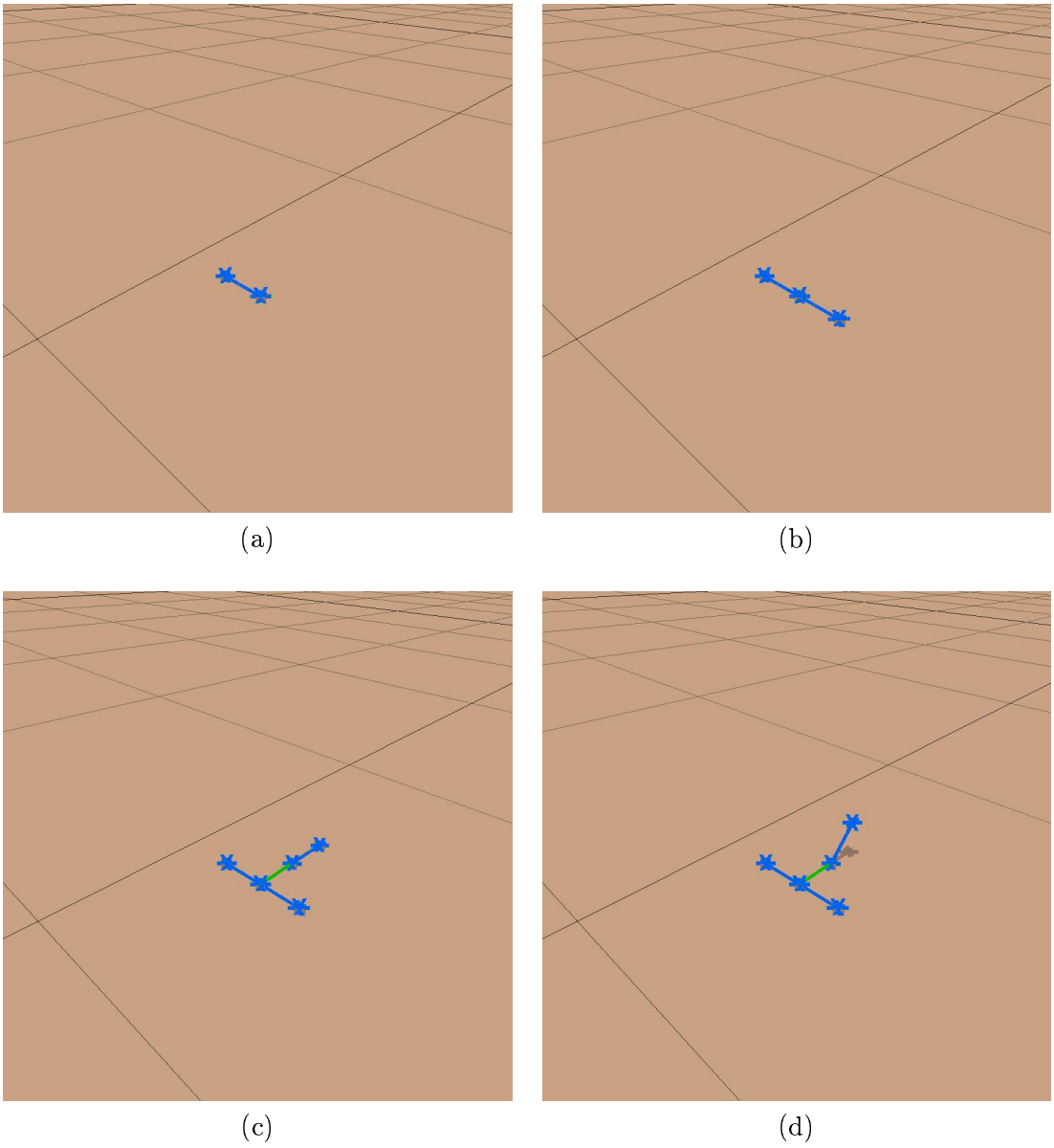
Figure 4.4: Basic building blocks of the system: rods of regular length and fixed and actuated joints.

turtle's heading about the appropriate axis in units of 90°. To create rods for robots, *forward* moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar and *back* goes backwards up the parent of the current bar. The joint commands move the turtle forward and end with a joint of the specified type which oscillates at a rate specified by the command's argument. By specifying the rate of oscillation and relative phase offset, a wide range of movement patterns can be generated. *Revolute-1(n)* creates a joint which oscillates from 0° to 90° about the Z-axis with speed n, *revolute-2(n)* creates a joint which oscillates from $-45°$ to 45° about the Z-axis with speed n, *twist-90(n)* creates a joint which oscillates from 0° to 90° about the X-axis with speed n, and *twist-180(n)* creates a joint which oscillates from $-90°$ to 90° about the X-axis with speed n. Combined with the rotation commands these allow actuated joints to be created that rotate about the primary axes. These commands are listed in table 4.5.

Figure 4.5 contains images of intermediate steps in building a genobot, as well as part

Table 4.5: Design language for oscillator-controlled, three-dimensional robots.

Command	Description
[ ]	*Push* and *pop* operators; they store/retrieve state to stack.
forward	Moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar.
back	Goes back up the parent of the current bar.
revolute-1(n)	Forward, end with a joint with range 0° to 90° about the current Z-axis and moves with speed n.
revolute-2(n)	Forward, end with a joint with range $-45°$ to 45° about the current Z-axis and moves with speed n.
twist-90(n)	Forward, end with a joint with range 0° to 90° about the current X-axis and moves with speed n.
twist-180(n)	Forward, end with a joint with range $-90°$ to 90° about the current X-axis and moves with speed n.
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.
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.
clockwise(n)	Rotate heading $n \times 90°$ about the turtle's X axis.
counter-clockwise(n)	Rotate heading $n \times -90°$ about the turtle's X axis.
increase-offset(n)	Increase phase offset by $n \times 25\%$.
decrease-offset(n)	Decrease phase offset by $n \times 25\%$.



(a) (b)

(c) (d)

Figure 4.5: Building and simulating a three-dimensional robot.

of its animation, from the following command sequence,

[left(1) forward] [right(1) forward] revolute-1(1) forward

The single bar in figure 4.5.a is built from the string, *[left(1) forward]*, and the two bar structure in figure 4.5.b is built from, *[left(1) forward] [right(1) forward]*. The final robot is made from the command sequence, *[left(1) forward] [right(1) forward] revolute-1(1) forward*, and is shown in figure 4.5.c, where it is displayed part-way through its movement cycle. Figure 4.5.d displays the robot with the actuated joint moved half-way through its joint range.

4.4 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 4.6, and the neural construction commands from section 4.2. The resulting language has two push/pop commands with two stacks: ( ), for pushing/popping the link-state to the link stack; and [ ], for pushing/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 a connection between body and brain is induced.

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 4.6. Before any commands are processed, a robot consists of a single oscil-

Table 4.6: Design language for neural-network controlled, three-dimensional robots.

Command	Description
[ ]	*Push* and *pop* operators; they store/retrieve state to stack.
forward	Moves the turtle forward in the current direction, creating a bar 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, end with a joint with range $0°$ to $90°$ about the current Z-axis that is controlled by the current neuron.
revolute-2	Forward, end with a joint with range $-45°$ to $45°$ about the current Z-axis that is controlled by the current neuron.
twist-90	Forward, end with a joint with range $0°$ to $90°$ about the current X-axis that is controlled by the current neuron.
twist-180	Forward, end with a joint with range $-90°$ to $90°$ about the current X-axis that is controlled by the current neuron.
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.
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.
clockwise(n)	Rotate heading $n \times 90°$ about the turtle's X axis.
counter-clockwise(n)	Rotate heading $n \times -90°$ about the turtle's X axis.
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.
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, 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.

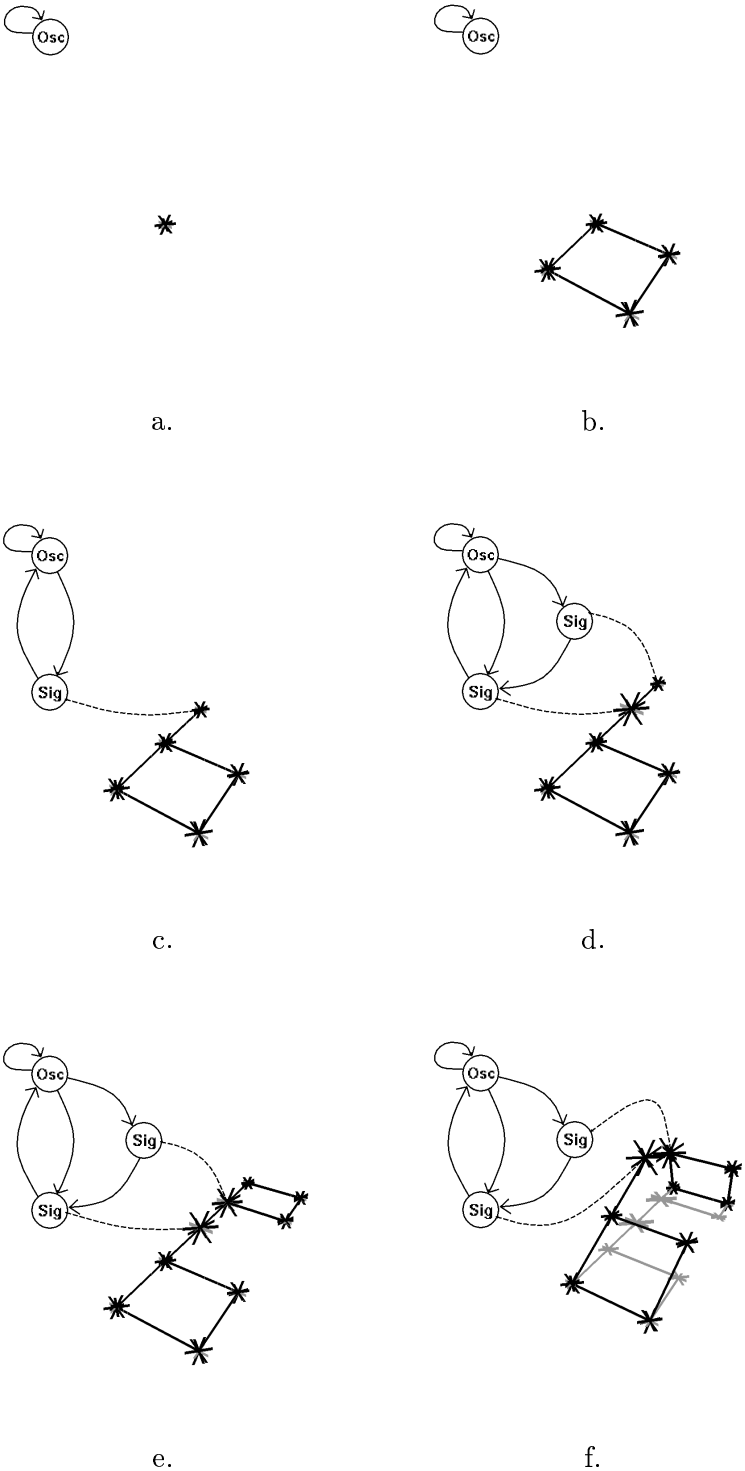


Figure 4.6: Constructing a neural-network controlled robot

lating neuron and a point, figure 4.6.a. 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 4.6.b. 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 4.6.c. The commands, *duplicate(0.25) split(0.4) reverse revolute-1(1.0)*, are repeated and a third neuron is created and it is attached to another actuated joint, figure 4.6.d. The last commands, *left(1.0) right(1.0) forward right(1.0) forward right(1.0) forward right(1.0) forward*, attach another square onto the end of the last revolute-1 joint, figure 4.6.e. Figure 4.6.f shows the creature with the joints halfway through their movement range.

4.4.1 Generative Representation Example for Neural Network Controlled Genobots

The following L-system consists of two production rules, each with two condition-successor pairs and combines the command sets for three-dimensional robots from section 4.3 with the neural network command set of section 4.2.

$$P0(n0): \quad n0 > 3.0 \rightarrow P1(5.0)\ P0(n0 - 2.0)\ left(1.0)\ P1(4.0)$$
$$n0 > 0.0 \rightarrow \{\ duplicate(0.25)\ split(0.4)\ reverse\ revolute - 1(1.0)\ \}(2.0)$$

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

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

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 }(4.0)*
4. *[{ right(1.0) forward }(4.0)] { duplicate(0.25) split(0.4) reverse revolute-1(1.0) }(2.0) left(1.0) { right(1.0) forward }(4.0)*
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 example from earlier in this section and the compiled design is shown in figure 4.6.

4.5 Robot Simulator

Once a string of construction commands is executed and the resulting robot is constructed, its behavior is evaluated in quasi-static kinematics simulator, similar to that of [88]. The kinematics are simulated by computing successive frames of moving joints in small angular increments of 0.001 radians toward the desired angle. This angle is determined by either an oscillator or a neuron. Oscillators cycle between -1 and 1 and this is scaled to the joint's range of motion. Similarly, the output value of a neuron falls within the range of -1 and 1 and this is scaled to the joint's range of motion. 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.

To achieve robot designs that are robust to transferal to the real world, error is added to evolved structures similar to the method of [70] and [63]. A robot design is evaluated by simulating it three times, once without error and twice with different error values applied

to 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 axis. 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.

4.6 Summary

In this chapter design domains for voxel structures, neural networks, oscillator controlled robots and neural network controlled robots were described and examples were given of how a design is constructed from an assembly procedure. These domains cover static and actuated physical structures, software and a combination of software and physical design thereby providing a general test suite for comparing design representations.

Chapter 5

Results

To determine if generative representations can capture design dependencies and are better able to explore large design spaces, a generative representation is compared against a non-generative representation (both described in chapter 3) on the four design substrates of chapter 4. In addition, the amount of reuse of the genotype with the generative representation is measured. First, the configuration of the two representations is given.

The non-generative representation was configured as an L-system with one production rule, no arguments, one condition-successor pair whose condition always succeeds, and without the repeat operator (iteration) or the ability to call production rules (abstraction). 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 or three condition-successor pairs, and two parameters for each production rule. For the generative representation, the maximum length of production body was set to fifteen commands and the maximum allowed length of a compiled generative representation was set to 10,000 commands – the same length as with the non-generative representation. The properties of both the generative and non-generative representations are listed in table 3.2.

The rest of this chapter presents the results of evolving designs with both a non-generative and a generative representation on the different classes of designs. Sections 5.1,

5.2, 5.3 and 5.4 present the results of comparing the non-generative and generative representations on designing tables, neural networks to solve even-parity, oscillator controlled robots and neural-network controlled robots. Examples of designs that were transferred to reality are shown in section 5.5 and these results are summarized in section 5.6.

5.1 Tables

The first class of design problems for which comparisons were made was the evolution of tables in a three-dimensional voxel world. The fitness of a table was a function of its: height, the maximum number of voxels above the ground; surface structure, the number of voxels at the maximum height; stability, and the volume of the table as calculated by summing the area at each layer of the table. Maximizing height, surface structure and stability typically result in table designs that are solid volumes, thus a measure of excess voxels is used to reward designs that use fewer bricks, which was the total number of voxels used in the design, not including those on the surface.

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

The resulting fitness function combined these measures into a single function[1],

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{penalty} \tag{5.1}$$

[1] A more appropriate method of evolving against these criteria may be to use a multi-objective approach [40].

The evolutionary algorithm was configured to run for 2000 generations using a population size of 200. After each generation the best two individuals were copied into the next population (an elitism of two) and the remaining individuals were created with an equal probability of using mutation or recombination. The grid size for the voxel world was 40 wide × 40 deep × 40 high.

5.1.1 Evolved Tables and Fitness Comparison

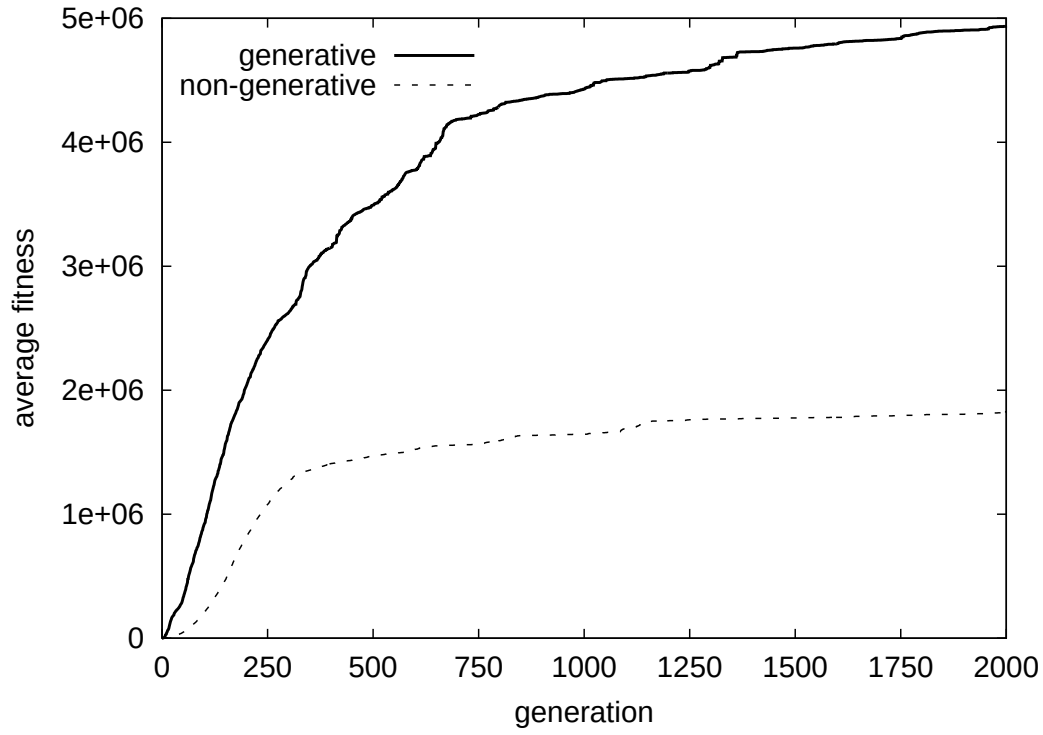
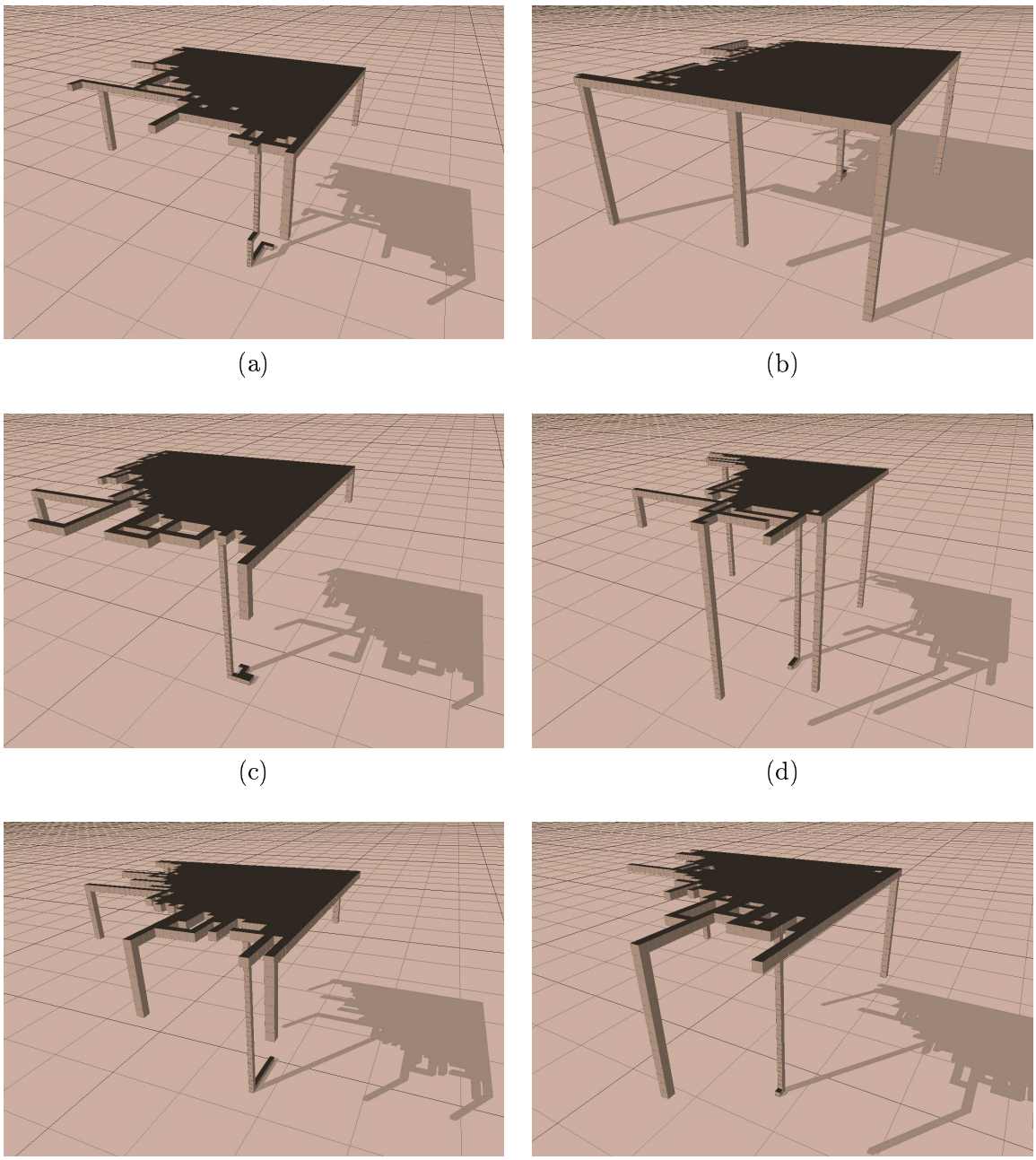


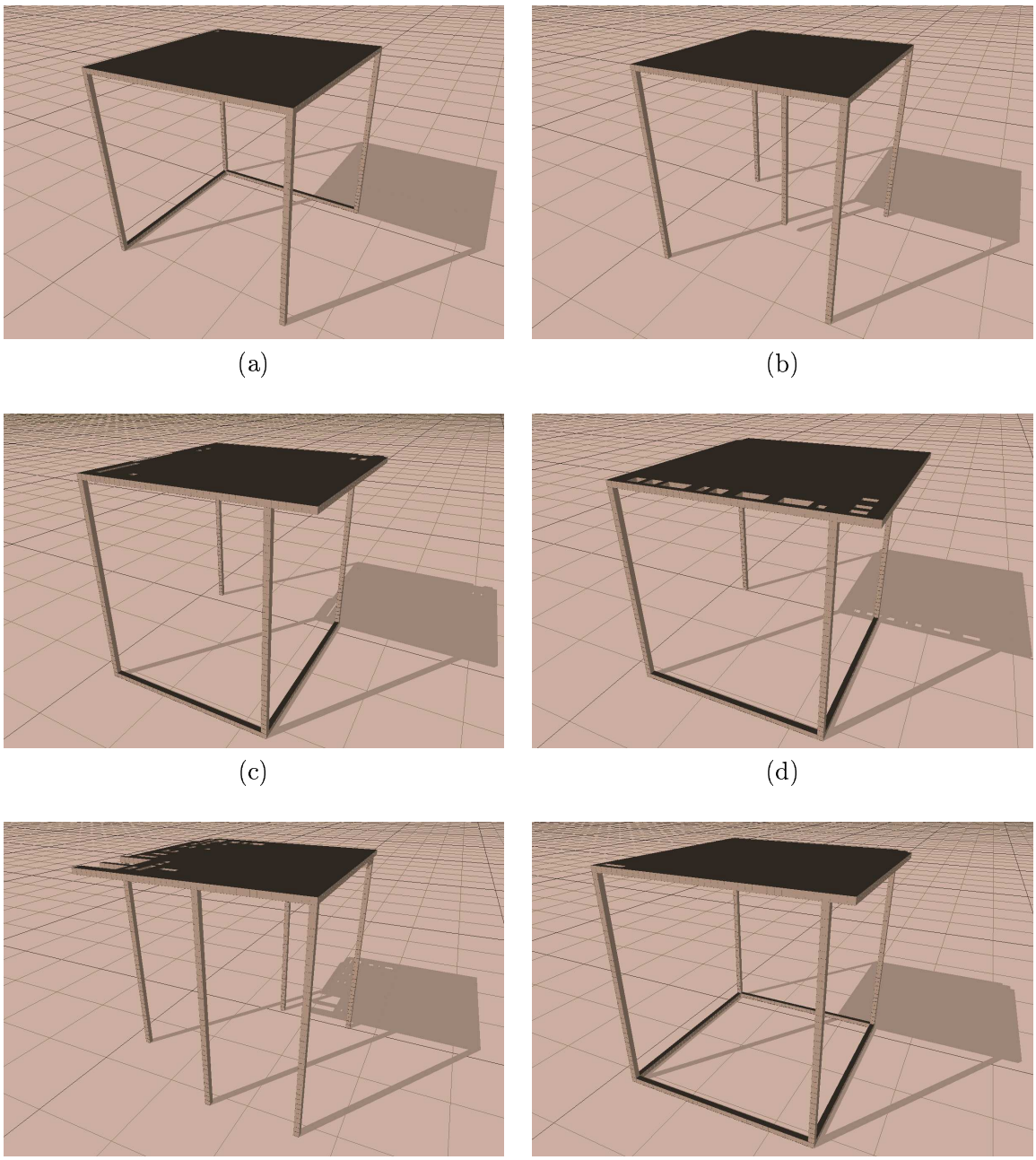
Figure 5.1: Fitness comparison between the non-generative and generative representations on evolving tables, averaged over fifty trials.

The graph in figure 5.1 contains a comparison of the fitness of the best individual evolved with the non-generative representation against the best individual evolved with the generative representation, averaged over fifty trials. With the non-generative representation, fitness improved rapidly over the first 300 generations, then quickly leveled off, improving by less than 25% over the last 1700 generations. Fitness increased faster with the generative



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

Figure 5.2: The best six tables evolved using the non-generative representation.



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

Figure 5.3: The best six tables evolved using the generative representation.

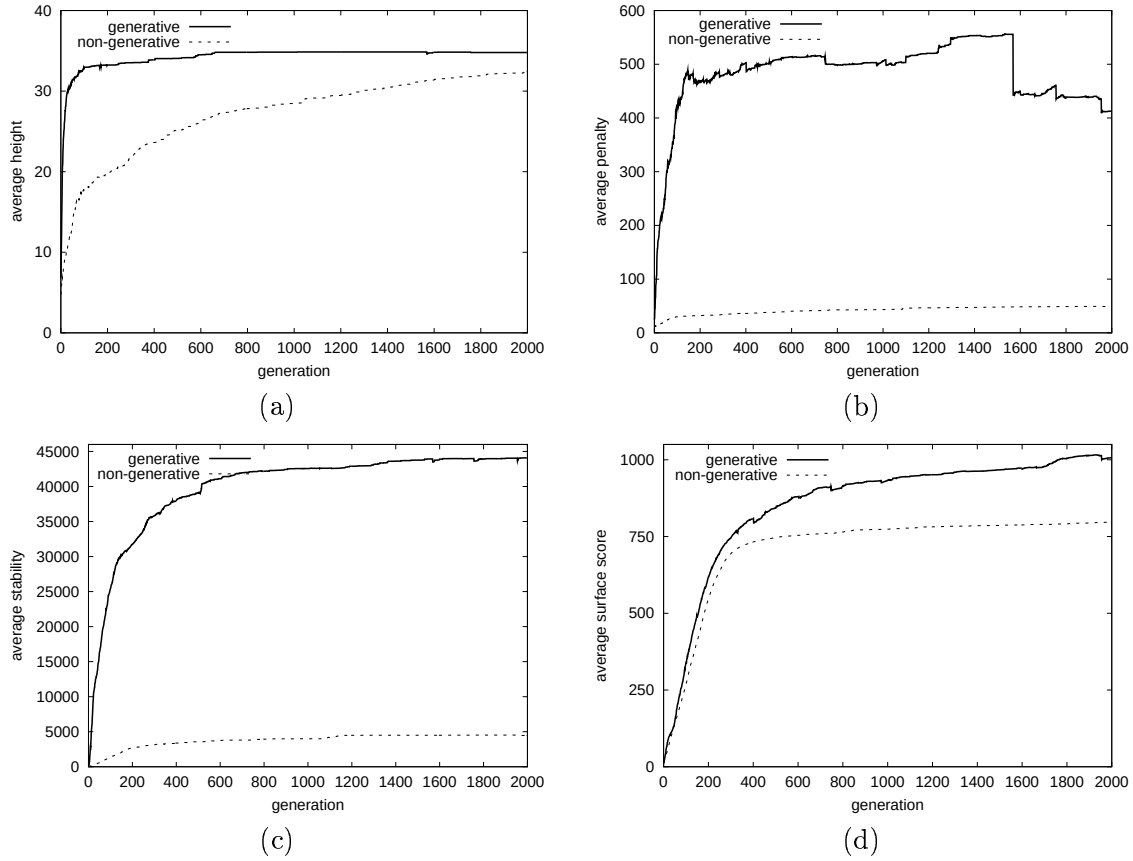


Figure 5.4: Graphs of: (a) height; (b) penalty; (c) stability; and (d) surface values against generation on the table design problem.

representation, and the rate of increase in fitness did not decrease as quickly as with the non-generative representation. The final results were an average best fitness of 1826158 with the non-generative representation and 4938144 with the generative representation.

Examples of the best tables evolved with each representation are shown in figures 5.2 and 5.3. The best tables evolved with the non-generative representation tend to be supported by only one table-leg, with parts of legs dropping from edges of the table-top. The likely reason for this is that it is not possible to change the length of multiple table-legs simultaneously with the non-generative representation, so the best designs (those with the highest fitness) had only one leg that raised the surface to the maximum height. The additional legs at the edge of the surface added to a table's fitness by contributing to the stability score.

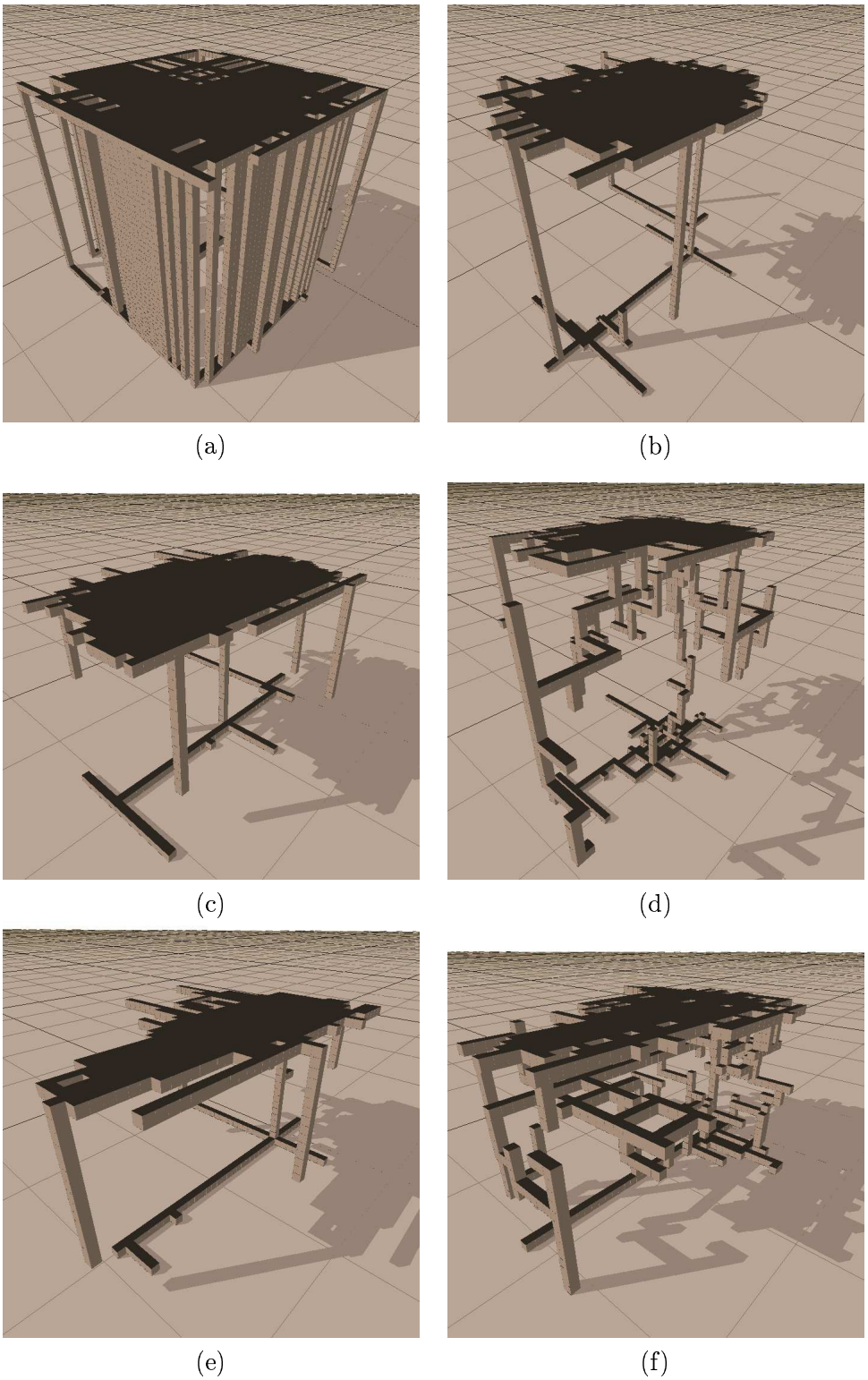
In contrast, the best tables evolved with the generative representation were supported by multiple legs that connected from the floor all the way to the table-surface.

The graphs in figure 5.4 show plots of the average height, penalty, stability and surface values for each generation. From these graphs it can be seen that with the non-generative representation table designs had small penalty and stability values, suggesting that there was only one table leg, which was increased over the generations. With the generative representation, designs had large penalty and stability values, suggesting multiple table-legs. Despite the dependencies of needing to adjust multiple table legs simultaneously, high tables were found quickly with the generative representation. It is probable, and evidence for this will be shown later, that the generative representation was reusing the same part of the genotype to create multiple table legs, allowing the height of these legs to be adjusted simultaneously with a single change to the genotype.

Using variations of the fitness function described in section 5.1, the tables shown in figure 5.5 are produced with the non-generative representation and the tables in figures 5.6 and 5.7 are tables evolved with the generative representation. In general, tables evolved with the non-generative representation were irregular and evolution with this representation tended to produce designs with few dependencies (most tables were supported by only one leg) whereas tables evolved with the generative representation had a reuse of parts and assemblies of parts and were supported with multiple-legs.

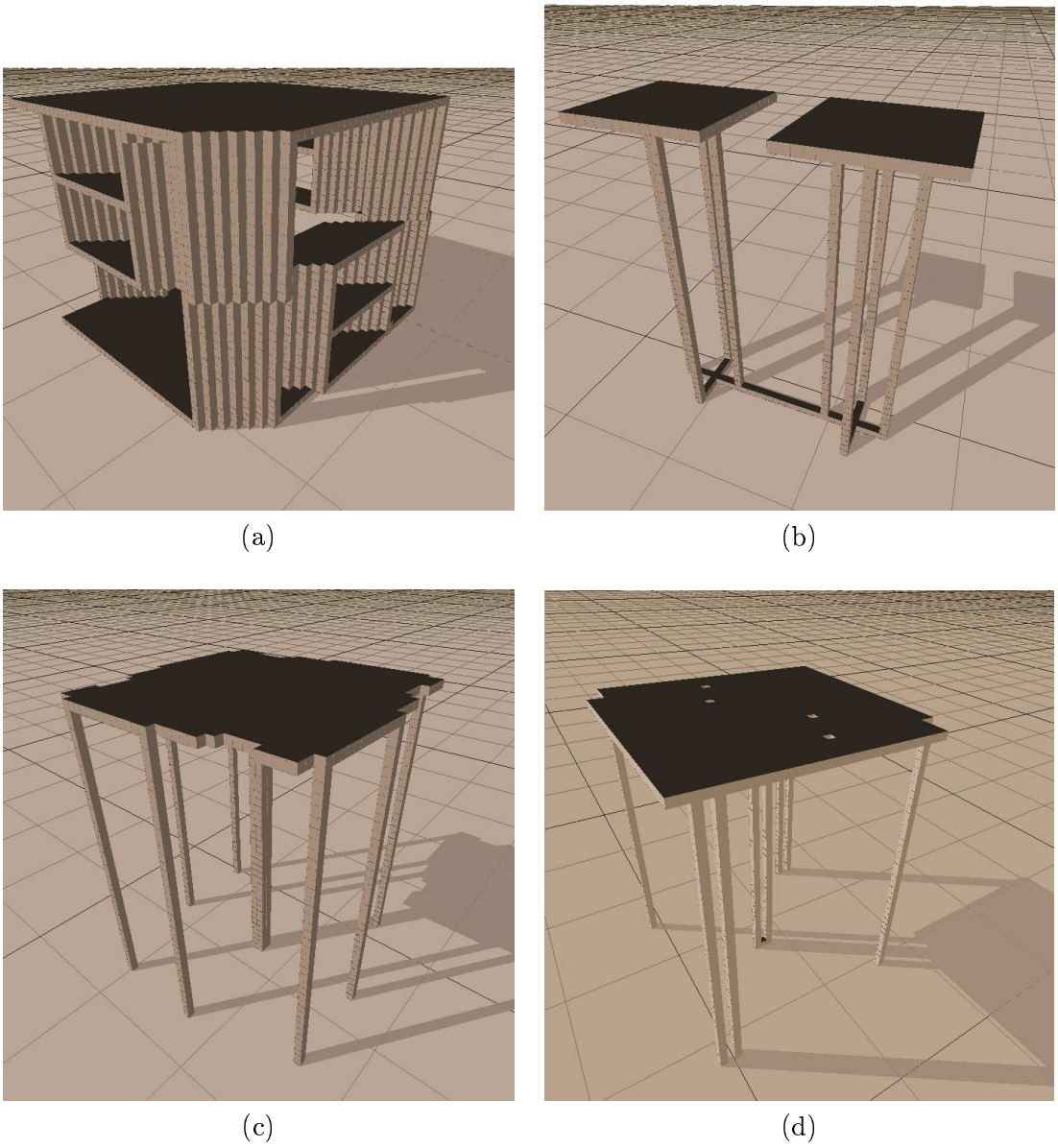
5.1.2 Reuse on the Table Design Problem

The difference between the generative representation and the non-generative representation is the ability for a generative representation to reuse parts of the genotype in creating the phenotype. The graph in figure 5.8.a plots the average length of the genotypes for both the non-generative and generative representations as well as plotting the average length of the assembly procedure produced by the generative representation. From this graph it can be seen that for the last 1250 generations the average length of encodings with the generative representation was approximately 310 symbols and the average length of the



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

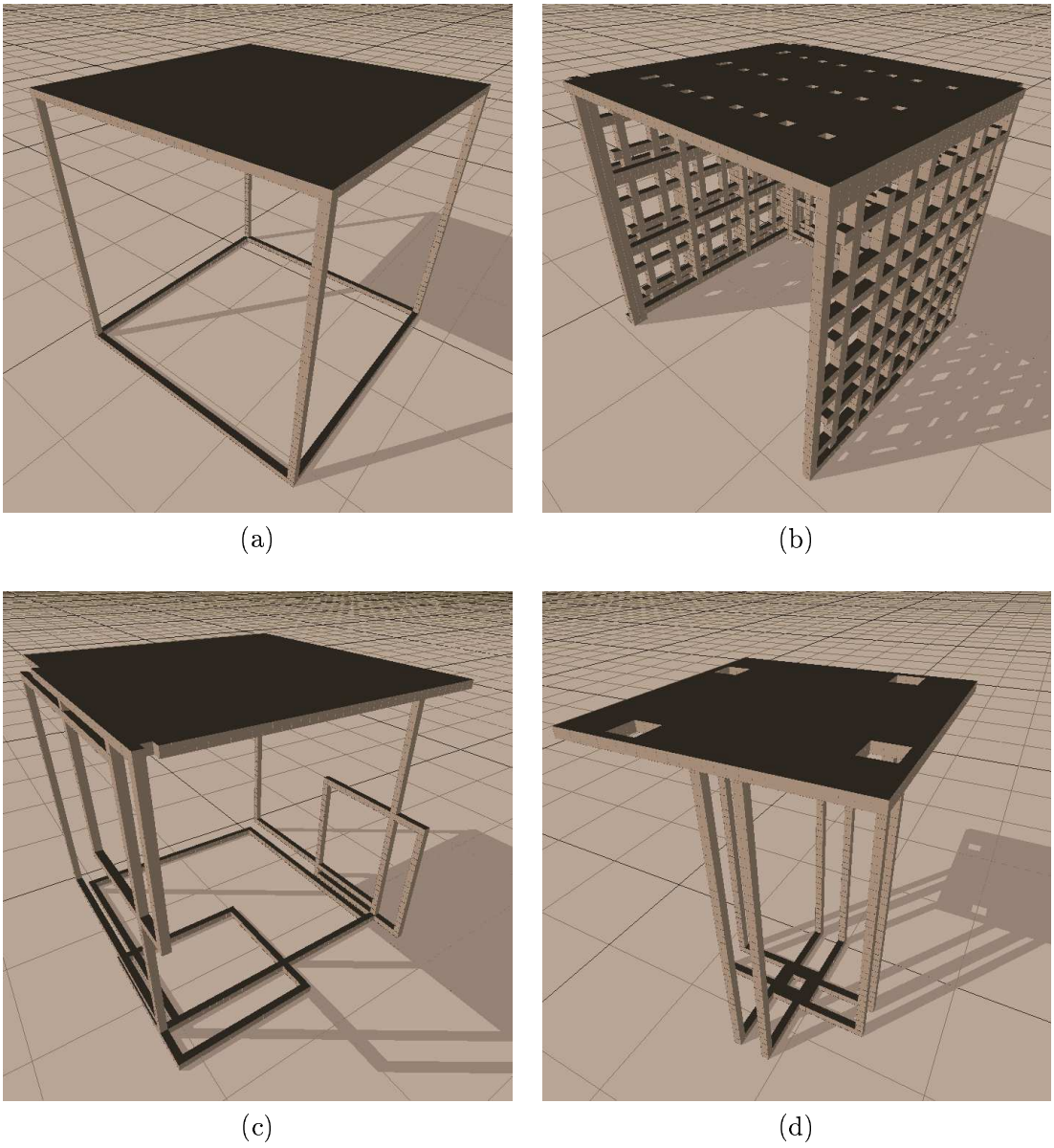
Figure 5.5: Other tables evolved using the non-generative representation.



(a) (b)

(c) (d)

Figure 5.6: Other tables evolved using the generative representation.



(a) (b)

(c) (d)

Figure 5.7: More tables evolved using the generative representation.

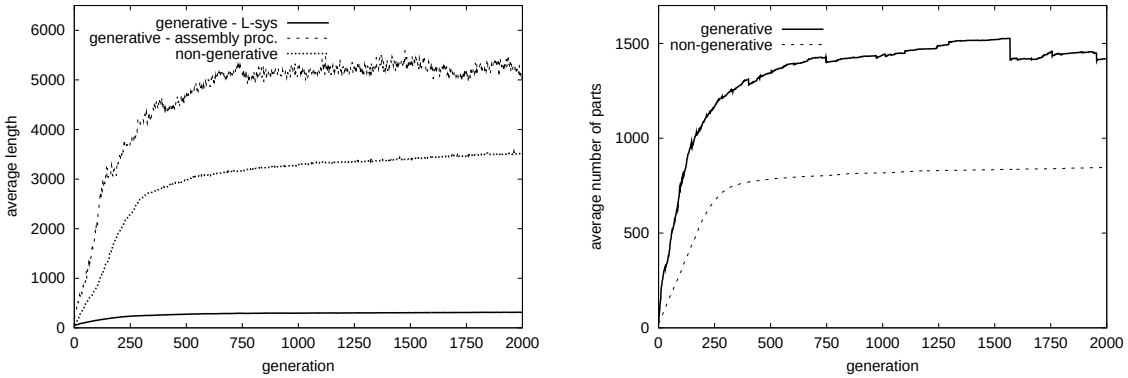


Figure 5.8: Graph of (a) length of the genotypes, and assembly procedure produced by the generative representation, against generation; and (b) graph of number of parts against generation.

assembly procedure it compiled to was about 5100 symbols long. Consequently parts of the genotype were being reused with the generative representation and the average number of times a symbol in the generative representation was reused in compiling to the assembly procedure was sixteen times.

It is possible that the generative representation was reusing parts of the genotype but that these reused symbols were not contributing to the final design (parts of the genotype that do not contribute to the phenotype are known as bloat [121]). Comparing the curves of the non-generative representation in both graphs shows a strong correlation between length of the genotype and number of parts in a design at a ratio of approximately four commands to one part. This ratio matches that of the construction language, in which two of the eight commands (forward and back) create a part. The curve with the generative representation's assembly procedure also has a similar ratio between length and number of parts suggesting that the reuse of the genotype was useful in that it resulted in the creation of parts.

5.1.3 Summary of Results for the Table Design Problem

In this section it was shown that evolutionary search found better designs with the generative representation than with the non-generative representation. Averaging the best fitness found

from fifty trials gives a score of 1826158 for the non-generative representation and 4938144 for the generative representation. In addition, with the generative representation, symbols in the genotype were used 16.1 times on average in creating assembly procedures.

5.2 Parity Solving Neural Networks

The second class of design problems used neural networks to solve the odd-n-parity function. The odd-n-parity function returns `true` if the number of `true` inputs is odd and returns *false* otherwise. This function is difficult because the correct output changes for every change of an input value. In addition, the even/odd-n-parity functions have become a standard benchmark function in genetic programming and past experiments have shown that GP does not solve the 5-parity (or higher) problem without automatically defined functions (a form of abstraction) [81].

Here recurrent neural networks were evolved to solve the odd-7-parity function. Input values were 1.0 for `true` and -1.0 for `false`. Networks were updated four times and then the value of output neuron was examined to determine the parity value calculated for that input. If the absolute value of the output neuron was > 0.9, the output of the network was taken as `true`/`false`. If the absolute value of the output neuron was < 0.9, the network was iteratively updated until its output value was > 0.9 or < -0.9, for a maximum of six updates. The network received a score of 2.0 for returning the correct parity value and a score of -1 for an incorrect answer. If the absolute value of the output neuron was less than 0.9 after all network updates, the network received a score of 1.0 if the value of the output neuron was positive and the parity was `true` or if the value of the output neuron was negative and the parity was `false` and no penalty was given for having an incorrect value in this case. The fitness value of a network was the sum of its scores on all possible inputs.

5.2.1 Fitness Comparison for 7-Parity Networks

The graph in figure 5.9 contains a comparison of the fitness of the best network evolved with the non-generative representation against the best network evolved with the generative representation, averaged over fifty trials. After 500 generations, the average fitness with the non-generative representation was approximately 190 and the average fitness with generative representation was approximately 230. For comparison, random guessing (both for all outputs less than 0.9 and for all outputs greater than 0.9) averages a score of 64, and the maximum score is 256.

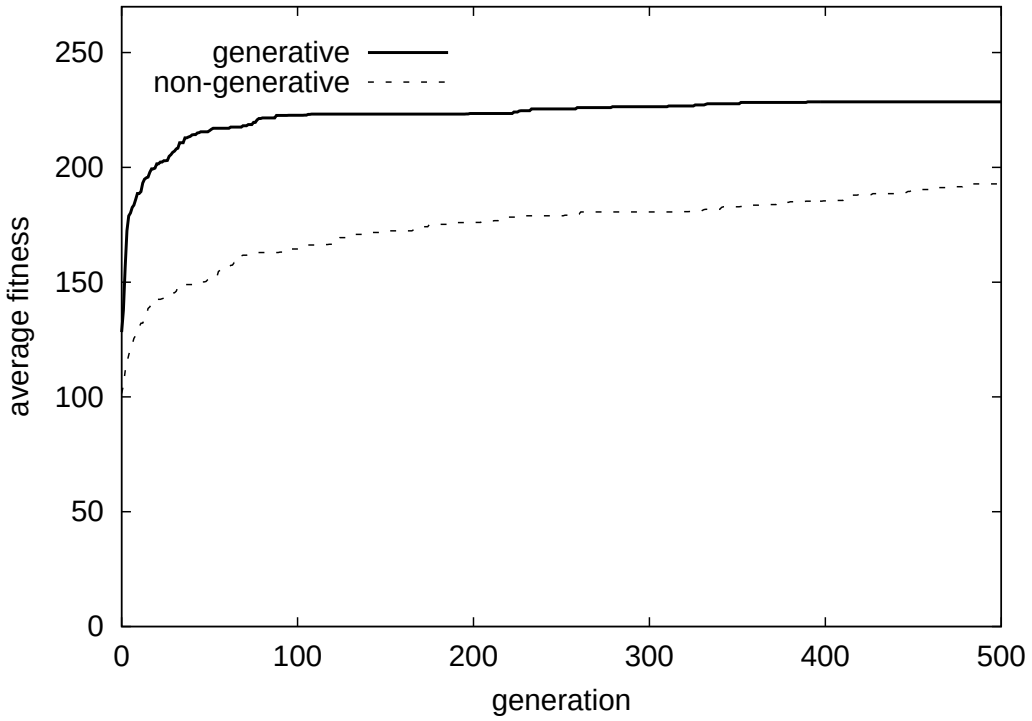


Figure 5.9: Fitness comparison between the non-generative and generative representations on evolving 7-parity networks.

Of the fifty runs with each representation, five runs with the non-generative representation and twelve runs with the generative representation produced networks which correctly calculated the 7-parity problem. The five correct networks with the non-generative representation are shown in figure 5.10 and the twelve correct networks with the generative

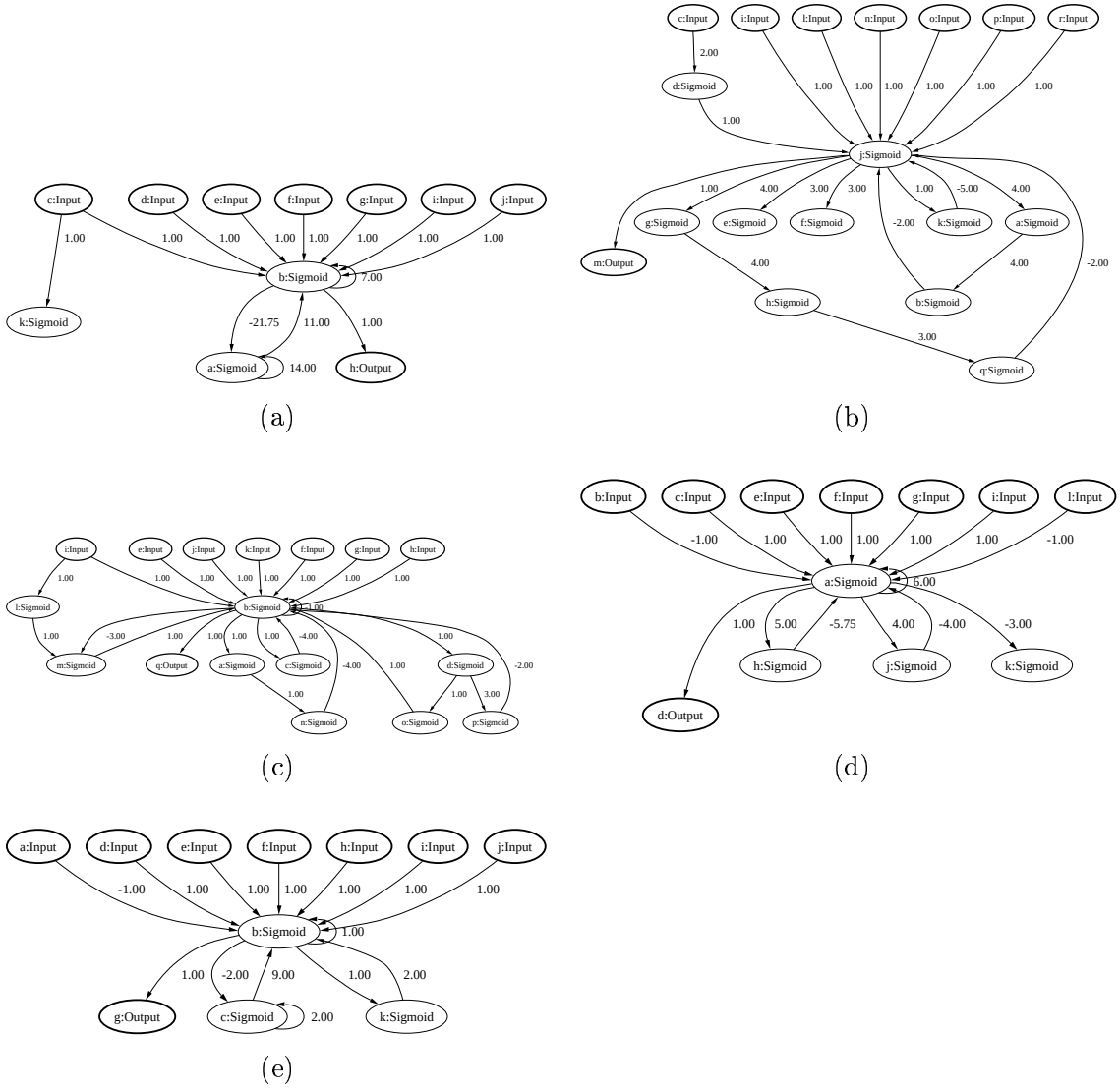


Figure 5.10: The five correct parity networks evolved with the non-generative representation.

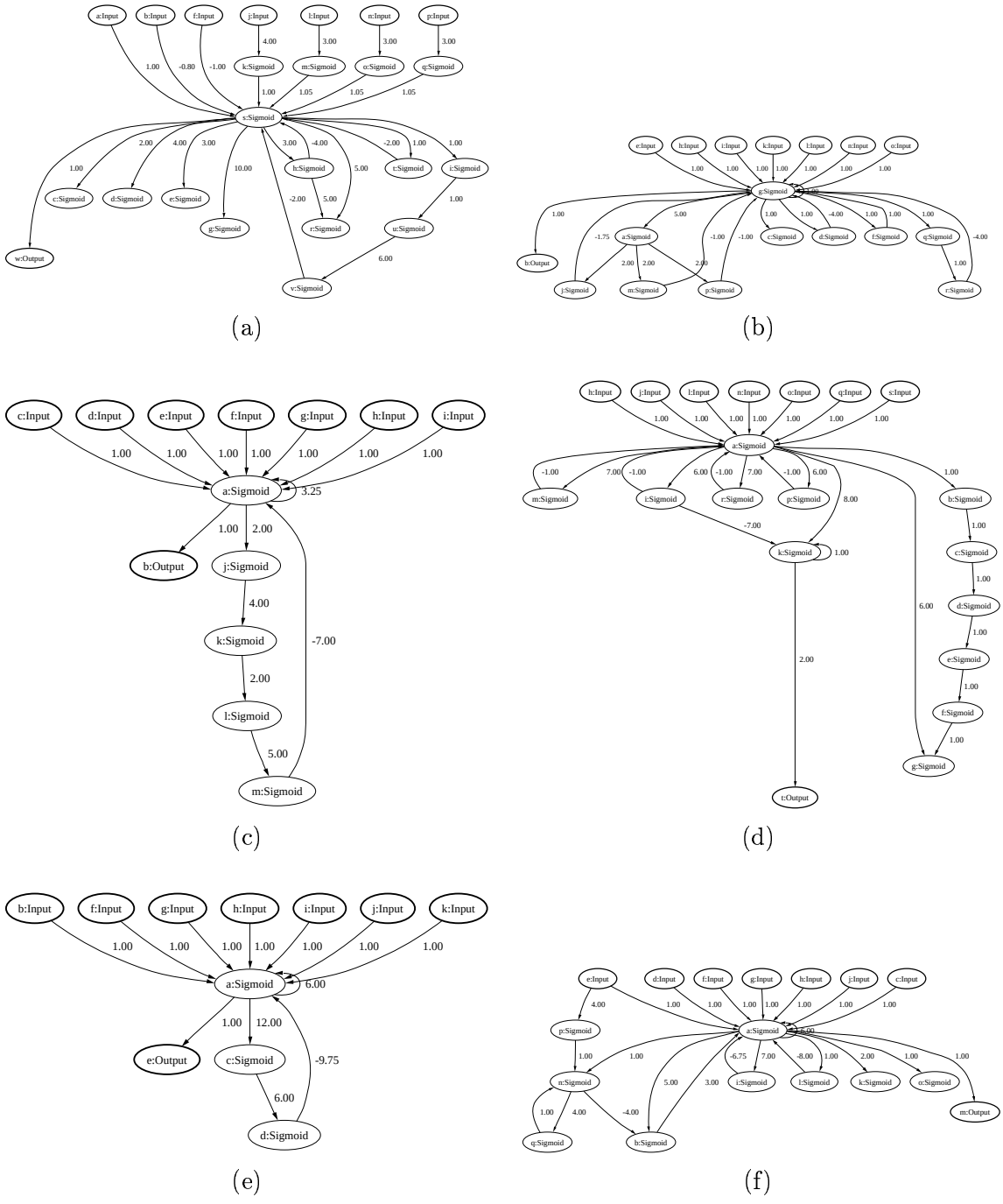


Figure 5.11: The first six correct parity networks evolved with the generative representation.

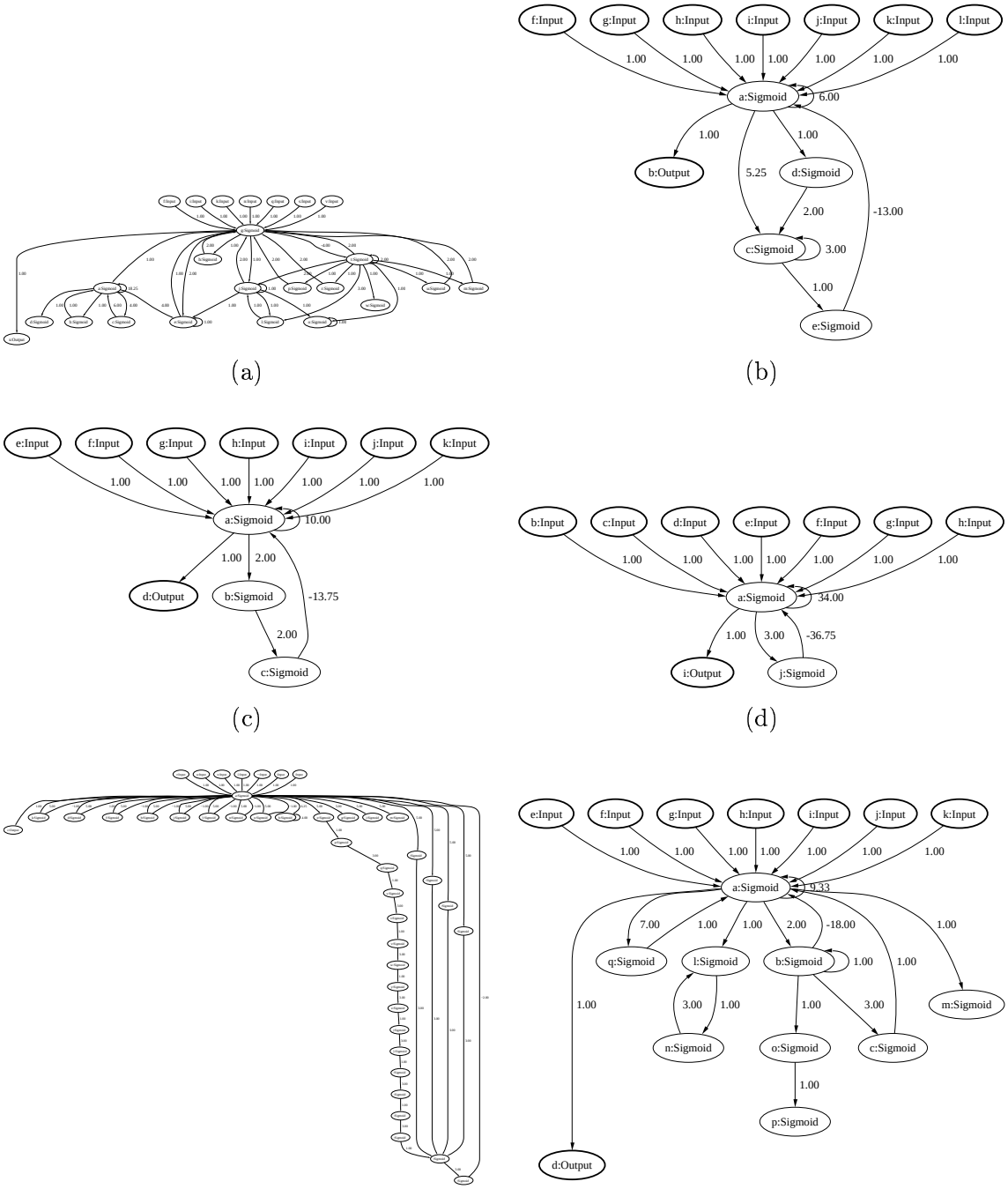
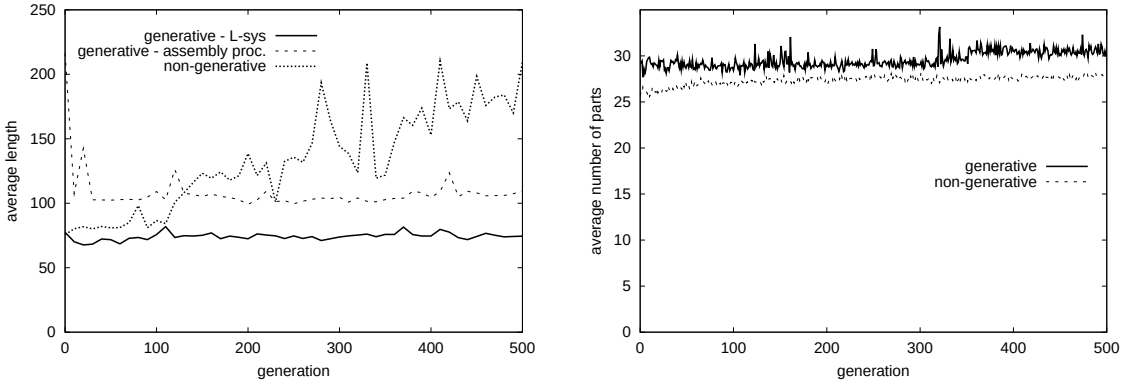


Figure 5.12: The second six correct parity networks evolved with the generative representation.

representation are shown in figures 5.11 and 5.12. Evolution with the generative representation found both the smallest and the largest correct networks. Because the evolved networks contain few parts, it is difficult to find much higher-order structure in the evolved networks.

5.2.2 Reuse for the 7-Parity Networks



(a) (b)

Figure 5.13: Graph of (a) length of the genotypes, and assembly procedure produced by the generative representation, against generation; and (b) graph of number of parts against generation.

The graph in figure 5.13.a plots the average length of the genotypes for both the non-generative and generative representations as well as plotting the average length of the assembly procedure produced by the generative representation. From this graph it can be seen that with the generative representation, data elements in the encoding of a parity network are used approximately one and a half times, on average, in creating an assembly procedure.

5.2.3 Summary of Results for the Parity Networks

The results of evolving networks to calculate parity are summarized in table 5.1. This table shows that the evolutionary algorithm was more likely to find correct networks, in fewer generations, with the generative representation than with the non-generative representation. This table also shows that the generative representation was using each symbol in the genotype 1.6 times, on average, in producing assembly procedures. Of the four classes of

Table 5.1: Summary of results for 7-parity networks.

Summary of results	Non-Generative	Generative
Average final best fitness	193	228
Percentage of runs which found solution.	10%	24%
Average generation solution was found (of runs that found a solution).	173	79
Average reuse of the genotype (final generation)	n.a.	1.6

designs this is the lowest degree of reuse and is likely because correct solutions did not need to be large (see figures 5.10, 5.11 and 5.12).

5.3 Oscillator Controlled Robots

The third class of design substrates on which the non-generative and generative representations were compared was that of designing robots in a simulated, three-dimensional environment. Using this design substrate the goal was to produce robots that moved across the ground as fast as possible. Fitness was a function of the distance moved by the robot's center of mass on a flat surface. In order to discourage sliding, 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 which would not function well in reality.[2] The non-generative representation was implemented as an L-system with one production rule, no arguments, 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, three condition-successor pairs, and two parameters for each production rule. For the generative

[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 then the one used here.

representation, the maximum length of 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 non-generative representation. The evolutionary algorithm used a population of 100 individuals and was run for 500 generations and results are the average over ten trials.

5.3.1 Evolved Oscillator-Controlled Robots and Fitness Comparison

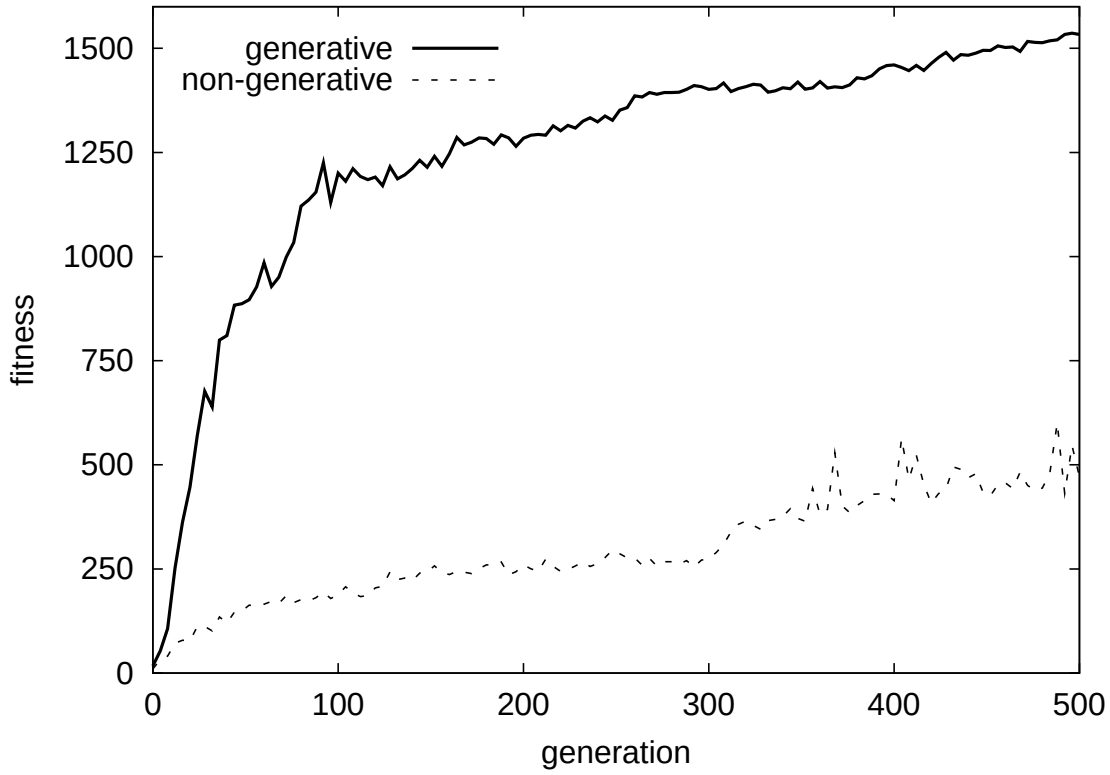


Figure 5.14: Performance comparison between the non-generative and generative representations on evolving robots with oscillator networks for controllers.

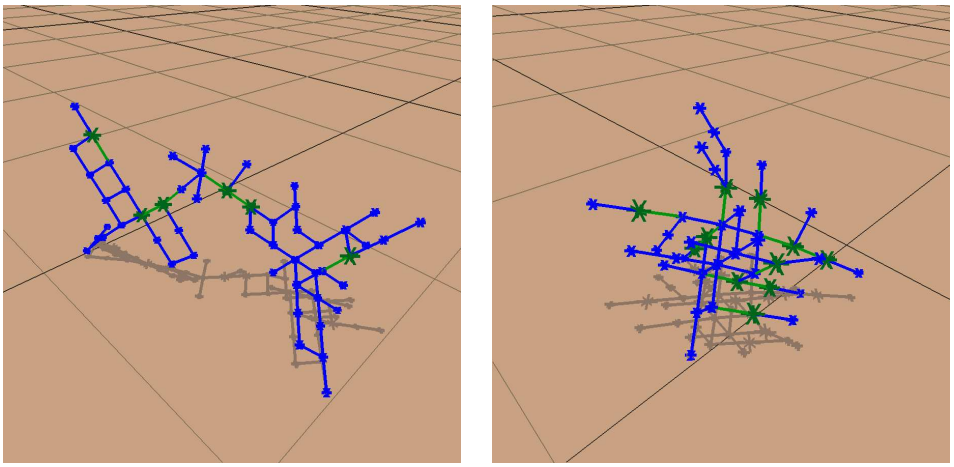
The graph in figure 5.14 plots the fitness of the best individual in the population, averaged over ten trials, for both the non-generative and generative representations. With the non-generative representation, the average fitness increased at a relatively constant rate from 0 to approximately 500. Using the generative representation, best fitness increased rapidly for

the first 100 generations, then slowed to a rate similar to the non-generative representation, and reached an average fitness value of just over 1500.

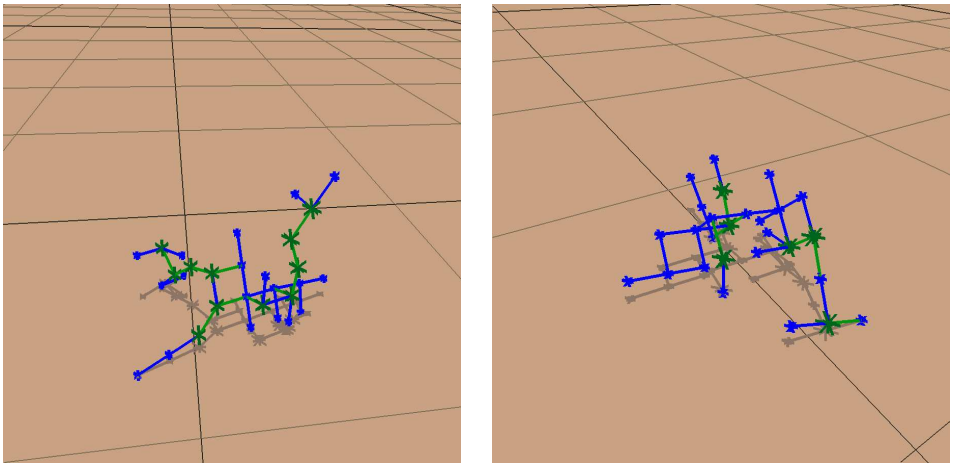
Evolutionary runs were similar in many ways. The first individuals started with a few rods and joints that would slowly slide on the ground. Robots produced from runs using the non-generative representation would improve upon their sliding motion over the course of the evolutionary run. The two main forms of locomotion found were using one, or more, appendages to push along or having two main body parts connected by a sequence of rods that twisted in such a way that first one half of the robot would rotate forward, then the other. The six fastest robots evolved with the non-generative representation are figure 5.15. The two fastest are figure 5.15.a (fitness 1188 with 49 rods and moves by twisting) and figure 5.15.b (fitness 1000 with 31 rods which moves by pushing). The robot in figure 5.15.c is an example of one that uses its appendages to roll over. Almost half the robots had only a handful of rods, such as the robot in figure 5.15.f, and these designs had the lowest fitness.

Genobots evolved with the generative representation not only had higher average fitness, but tended to move in a more continuous manner. The six fastest evolved genobots are in figure 5.16. Of these, the two fastest are the genobot in figure 5.16.a, whose segments are shaped like a coil and it moves by rolling sideways with fitness of 3604 and 325 rods, and the genobot in figure 5.16.b, a sequence of interlocking X's that rolls along with fitness 2754 and 268 rods. Not all evolved genobots showed a reuse of assemblies of components, as demonstrated by the ones in figure 5.16.d and f. An example of the movement cycle of a robot produced with the generative representation is in figure 5.17. This genobot, constructed from 80 rods with a fitness of 686, moves by passing a loop from back to front. In general, genobots evolved using the generative representation increased their speed by repeating rolling segments to smoothen out their gaits, and increasing the size of these segments/appendages to increase the distance moved in each oscillation.

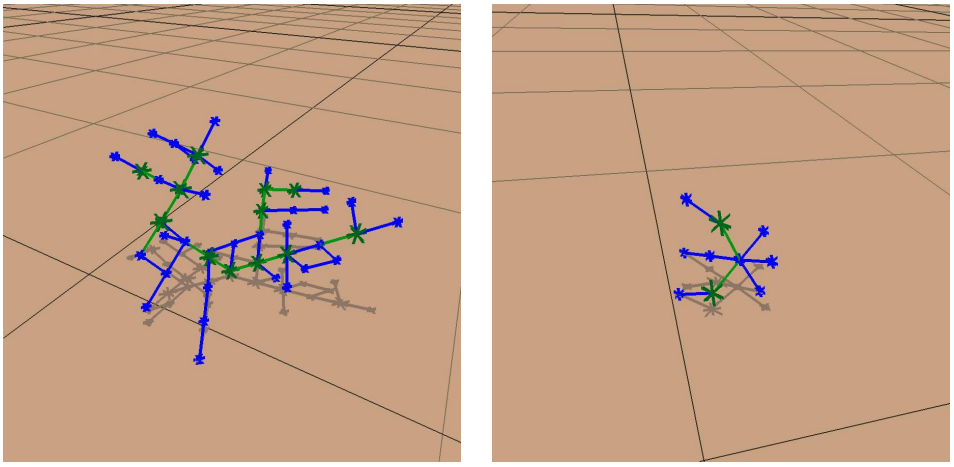
Additional runs with the non-generative encoding to evolve both oscillator and neural-network controlled robots failed to yield designs more interesting than those in figures 5.15 and 5.21. In contrast, additional runs with the generative representation produced a variety



(a) (b)

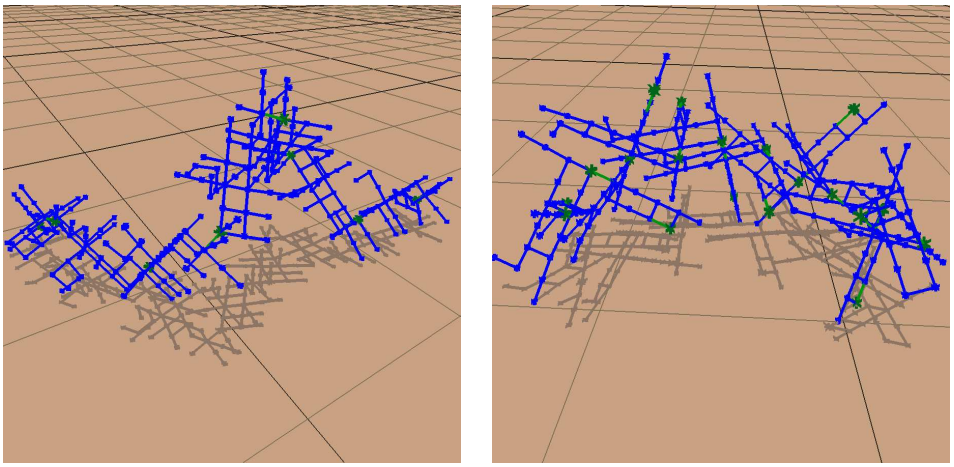


(c) (d)



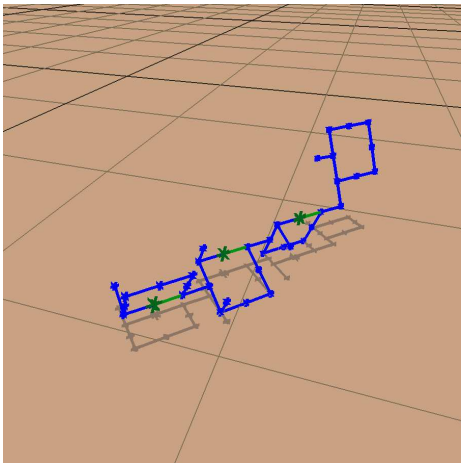
(e) (f)

Figure 5.15: The best six oscillator controlled robots evolved using the non-generative representation.

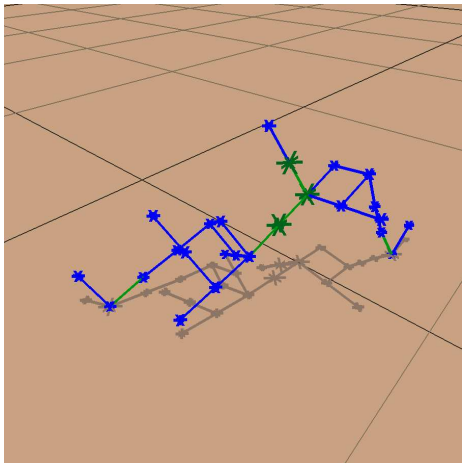


(a)

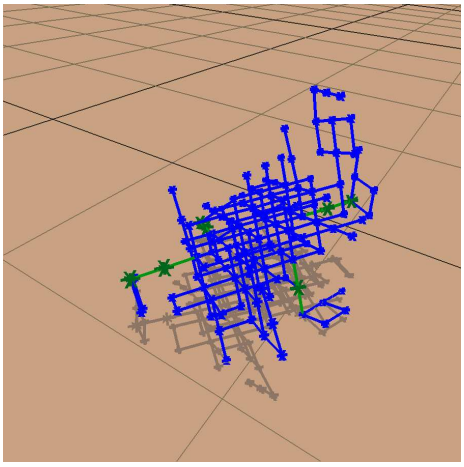
(b)



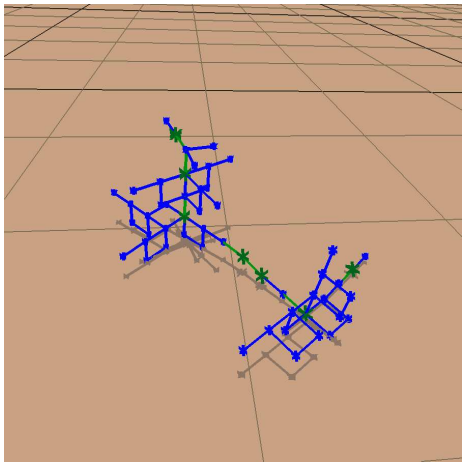
(c)



(d)



(e)



(f)

Figure 5.16: The best six oscillator controlled genobots evolved using the generative representation.

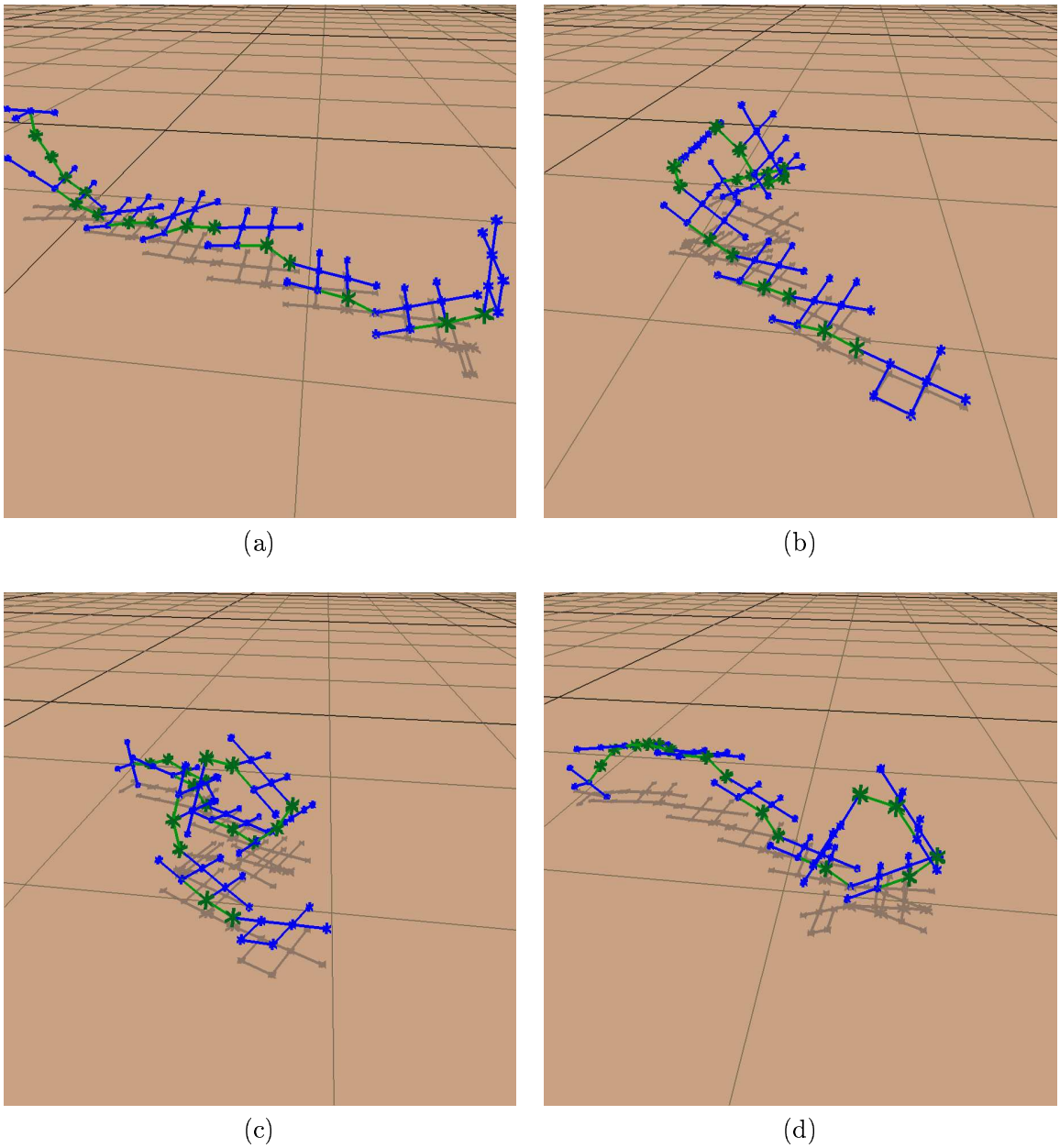


Figure 5.17: Part of the locomotion cycle of an oscillator-controlled genobot.

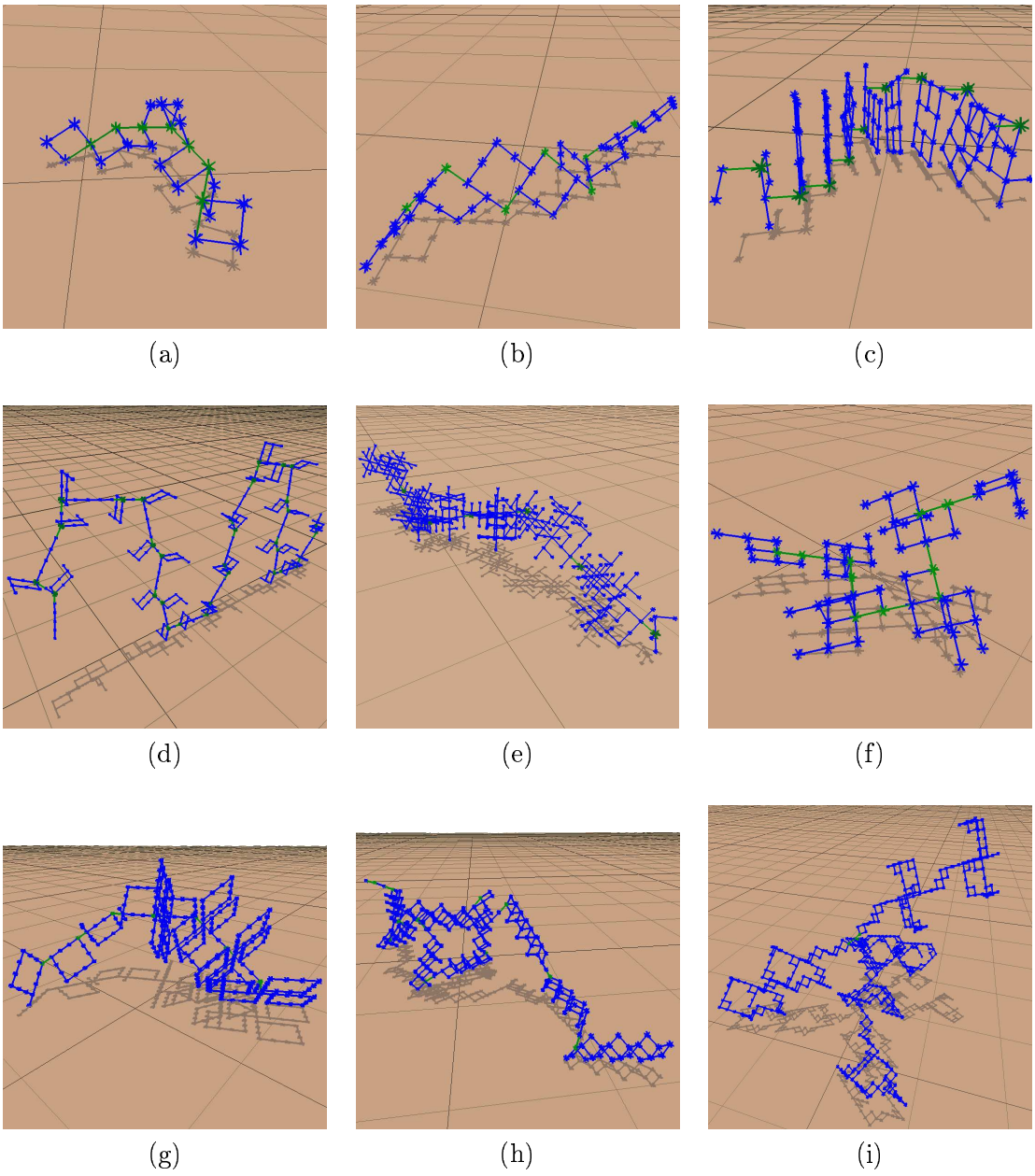


Figure 5.18: A variety of evolved 3D oscillator robots.
These consist of: *a*, a rolling genobot with 33 bars; *b*, a bi-connected rolling chain with 59 bars; *c*, a sequence of rolling rectangles with 169 bars; *d*, an undulating serpent with 339 bars; *e*, a 5-segmented inch-worm with 414 bars; *f*, a flipping genobot with 99 bars; *g*, an asymmetric rolling genobot with 306 bars; *h*, a coiling snake-like genobot with 342 bars; and *i*, a four-legged walking genobot with 629 bars.

of genobots with different styles of locomotion. The most common form of movement for evolved oscillator-controlled genobots was to roll along sideways, as done by the chains in figures 5.18.a, 5.18.b, 5.18.c and 5.18.g. The creature in 5.18.d moves like an undulating sea-serpent and, in a similar way, the creature in figure 5.18.e moves like an inch-worm. Another common form of locomotion, similar to rolling, is the flipping of the creature in figure 5.18.f. Instead of performing a continuous rotation of its body, this creature repeatedly moves its center of mass outside its contact points and falls over. Two of the larger creatures that evolved are the one in figure 5.18.h, which moves by pushing a coil from front to back and then re-creating the coil at its front, and the creature in figure 5.18.i which uses four legs in an awkward walk.

5.3.2 Reuse with Oscillator-Controlled Robots

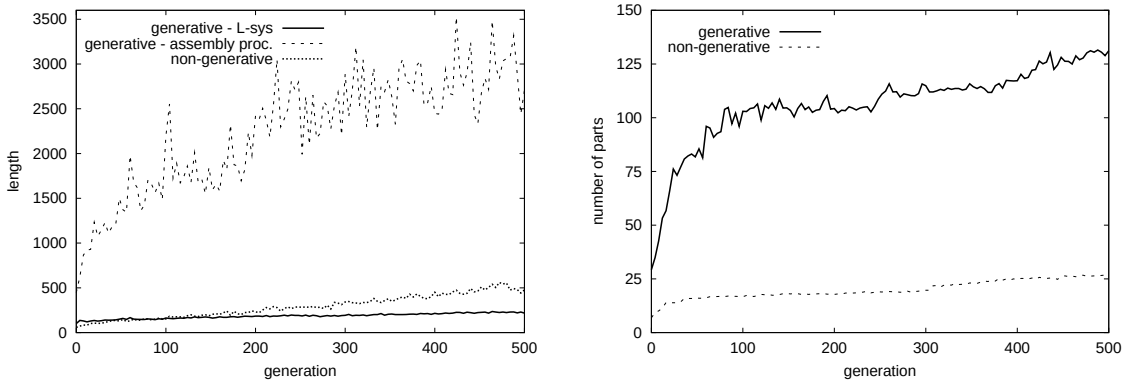


Figure 5.19: Graph of (a) length of the genotypes, and command string produced by the generative representation, against generation, and (b) graph of number of parts against generation.

The reuse of parts by the generative representation is shown in the graphs in figure 5.19. This graph contains plots of the average length of the most fit genotype for both representations as well as the average length of the assembly procedure produced from the generative representation. These plots show that with the generative representation the genotype increased from 100 symbols to 220 and its assembly procedure increased from 450 symbols to

3000 symbols – a change in symbol reuse from 4.5 to 13.6. Comparing the average number of parts in designs of both representation is one way of determining whether this reuse of genotype was beneficial in the final design, or just a form of bloat. The graph in figure 5.19.b plots the average number of parts for the best designs with the two representations. Comparing the average number of parts with the average length of the assembly procedure for both representations shows that at generation 500, there were approximately 17 commands/part with the non-generative representation's assembly procedures, 1.7 commands/part with the generative representation and 23 commands/part with the generative representation's assembly procedure. While the non-generative representation had a more efficient ratio of assembly-procedure to number-of-parts, the generative representation is more efficient by an order of magnitude in genotype-length:number-of-parts.

5.3.3 Summary of Results for Oscillator-Controlled Robots

In summary, better robots were evolved using the generative representation than with the non-generative representation. Averaged over ten trials, the best fitness found with the non-generative representation was 471 and the best fitness found with the generative representation was 1533. Also, designs encoded with the generative representation used symbols 12.4 times, on average, in translating from the genotype to the assembly procedure.

5.4 Neural Network Controlled Robots

This fourth design substrate combines the evolution of neural networks with the evolution of robot morphology for the task of locomotion. The fitness function was the same as that used for the oscillator-controlled robots and is described in section 5.3. The non-generative representation was implemented as an L-system with one production rule, no arguments, 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 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 non-generative representation. Implementing the non-generative 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 non-generative representation and ten with the generative representation.

5.4.1 Evolved Neural Network-Controlled Robots and Fitness Comparison

The first graph, figure 5.20, contains plots of the best fitness (averaged over ten trials) of the best individuals evolved with the non-generative representation and the generative representation. After ten generations the generative representation achieved a higher average fitness than runs with the non-generative representation did after 250 generations and the final genobots evolved with the generative representation were more than ten times faster, on average, than robots evolved with the non-generative representation.

Figure 5.21 shows the six best individuals evolved with the non-generative representation and figure 5.22 shows the six best genobots evolved with the generative representation. From the images it can be seen that the robots evolved with the non-generative 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.

As with oscillator-controlled robots, further runs with the generative representation produced robots with different styles of locomotion but failed to produce new varieties of robots with the non-generative representation. The images in figure 5.23 are examples of other

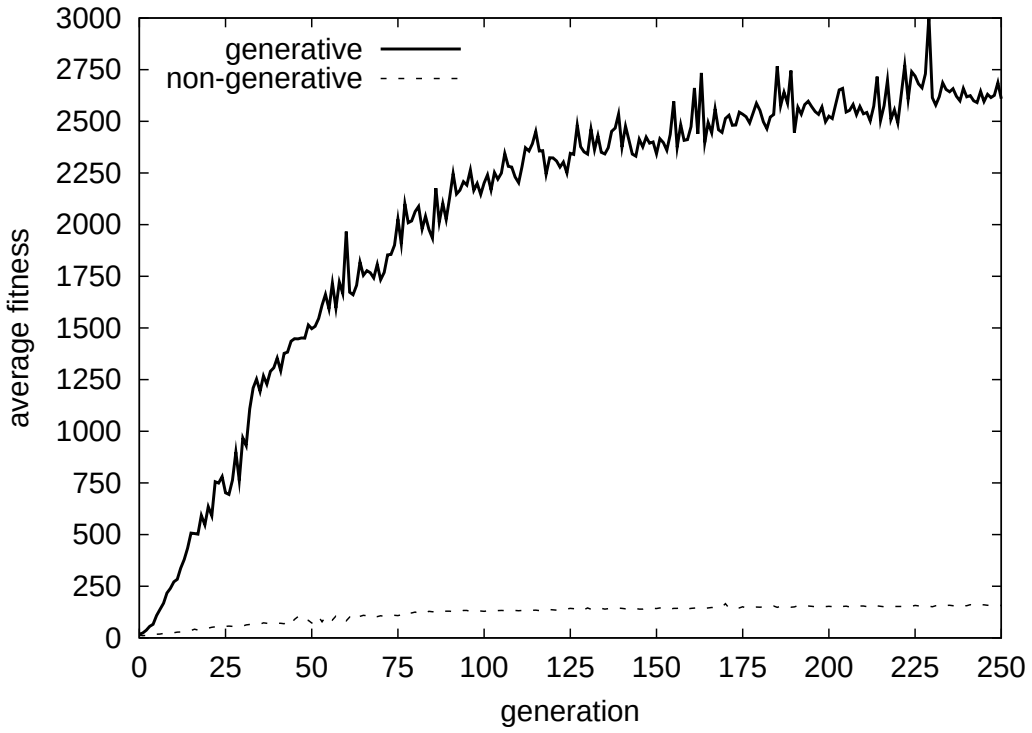


Figure 5.20: Performance comparison between the non-generative representation and the generative representation on evolving robots with neural networks for controllers.

robots evolved with the generative representation in which there was no constraint requiring cycles on limbs longer than four rods. The robot in figure 5.23.a is connected almost entirely of actuated joints. It moves by alternating between pulling all its limbs in tight to its body and extending them while twisting its torso. A kind of wheel was produced in one run, shown by the robot in figure 5.23.b. It moves by using its tail to continually turn its body over and over. The robot in figure 5.23.c is similar to early versions of the undulating serpent of figure 5.18.d with articulated joints between body segments for a kind of inchworm-like motion. Finally, the robot in figure 5.23.d is another example of a rolling chain of segments which was commonly evolved for both oscillator and neural-network controlled robots.

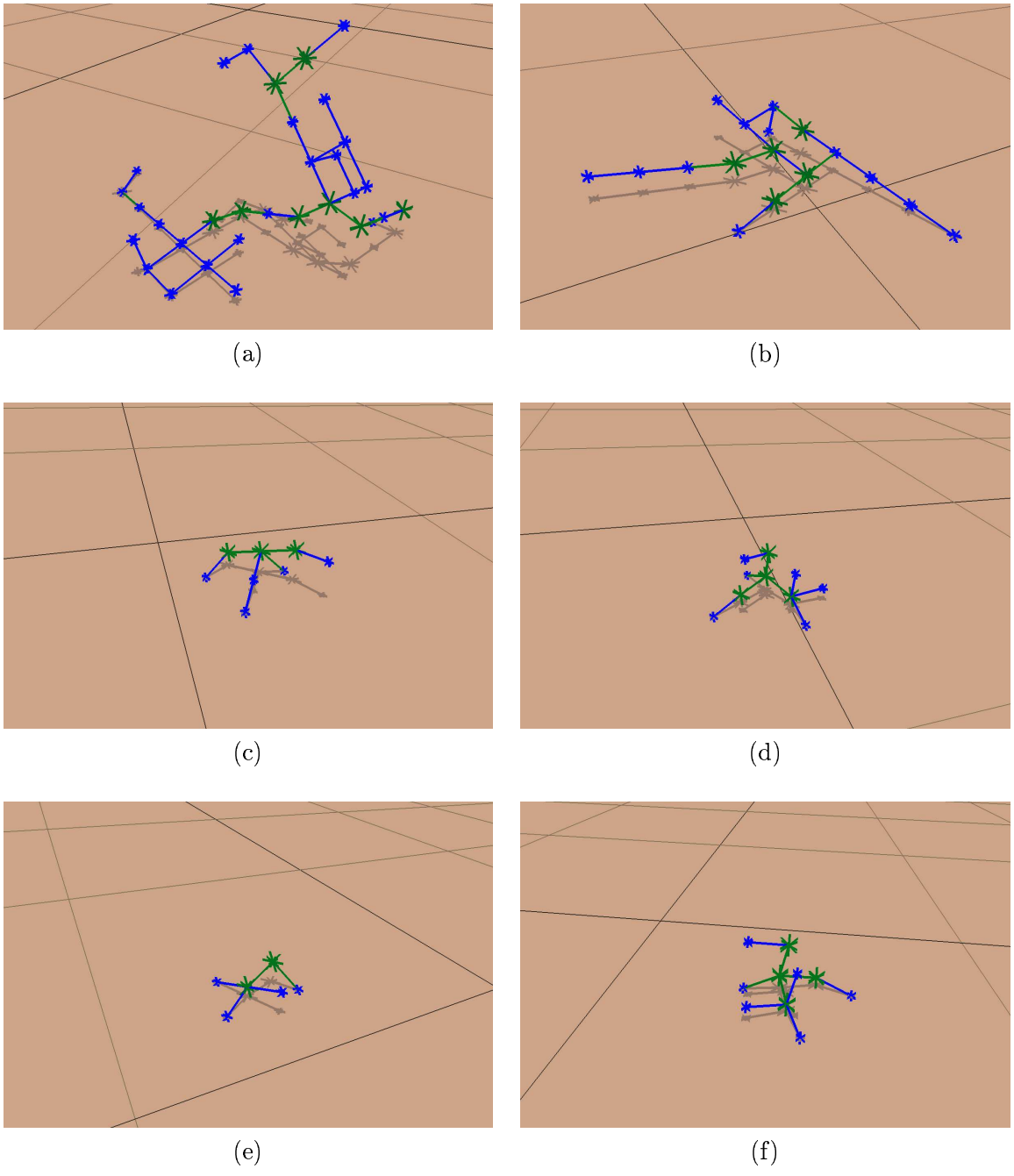


Figure 5.21: The best six neural-network controlled robots evolved with the non-generative representation.

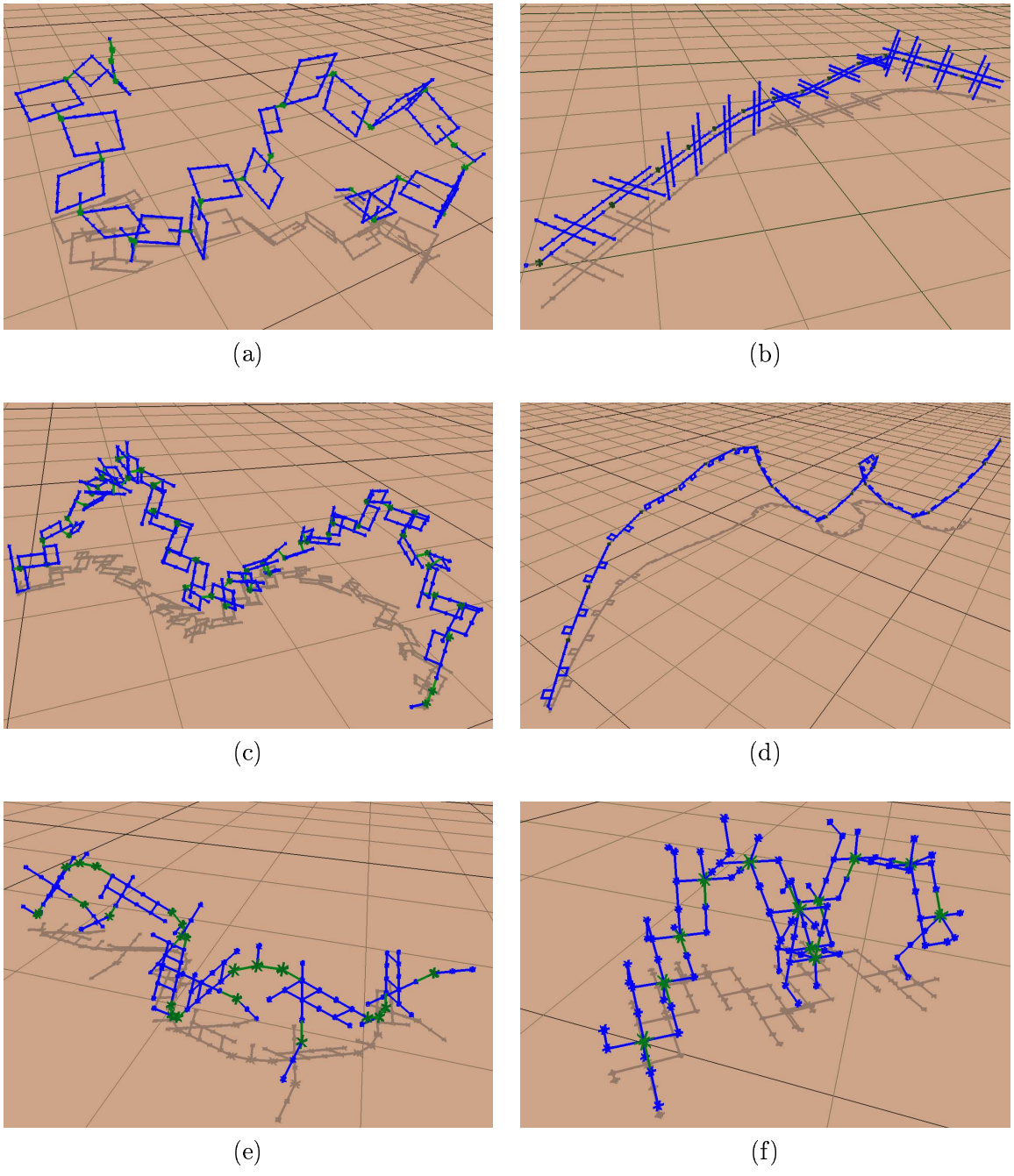


Figure 5.22: The best six neural-network controlled genobots evolved with the generative representation.

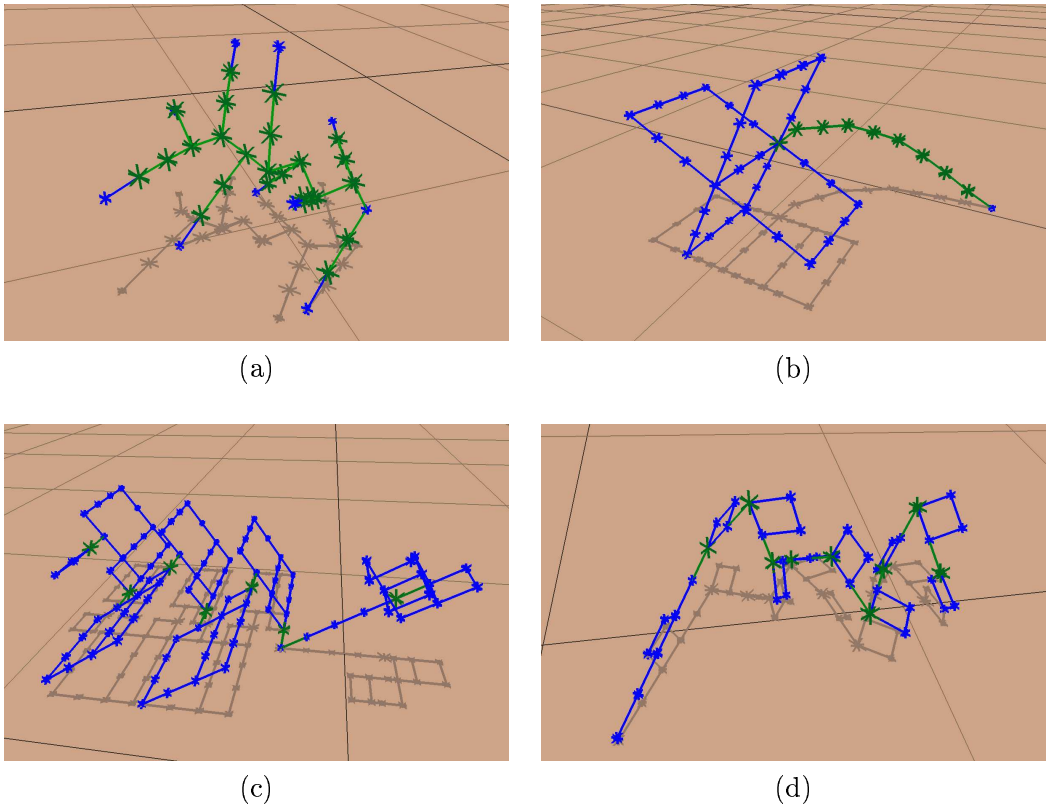


Figure 5.23: Other genobots evolved using the generative representation on runs with no constraints on limb lengths.

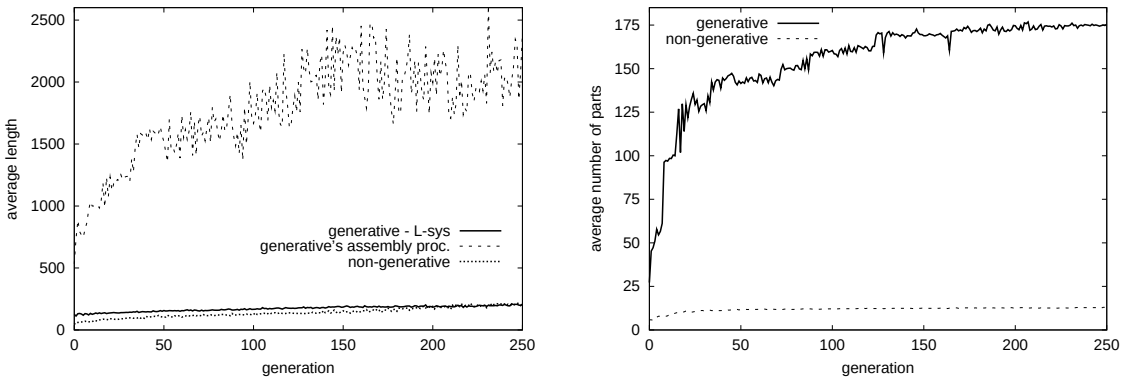


Figure 5.24: Graph of (a) length of the genotypes, and command string produced by the generative representation, against generation, and (b) graph of number of parts against generation.

5.4.2 Reuse for the Network-Controlled Robots

In the introduction it was argued that one advantage of a generative representation is its ability to better search large design spaces through reuse of elements in an encoded design. The graph in figure 5.24.a 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 a 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 2387 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 5.24.b. As the genotype sizes for both non-generative and 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 comes from the images in figures 5.23.c-h, which show that designs evolved with the generative representation have the same assemblies of parts occurring multiple times in a genobot.

In addition to reuse of components in a genobot's morphology, the networks constructed from the generative representation also contain reuse of parts. The neural network controller shown in figure 5.25 is the controller for the genobot in figure 5.23.h, and it shows a two-layer hierarchy of reuse. First, the two-node subnetwork of a sigmoid unit, with a link to itself and which is connected to an output unit, is repeated multiple times. Second, a link connecting every second one of these subnetworks shows that higher-level assembly has been created on top of the first. As well as demonstrating reuse of components, its linear sequence of outputs also corresponds to the linear sequence of joints in the genobot's morphology suggesting that the dependency between the genobot's controller and morphology are captured in the encoded robot design.

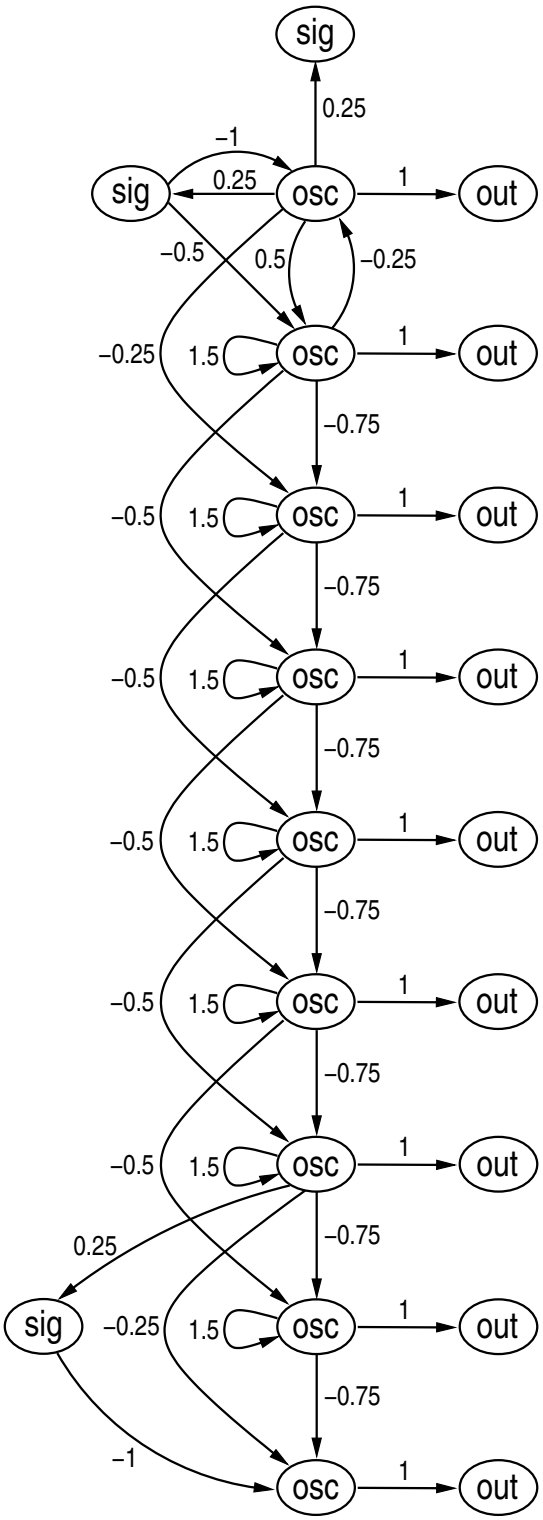


Figure 5.25: Evolved neural network controller for the genobot in figure 5.23.d.

5.4.3 Summary of Results for the Neural Network-Controlled Robots

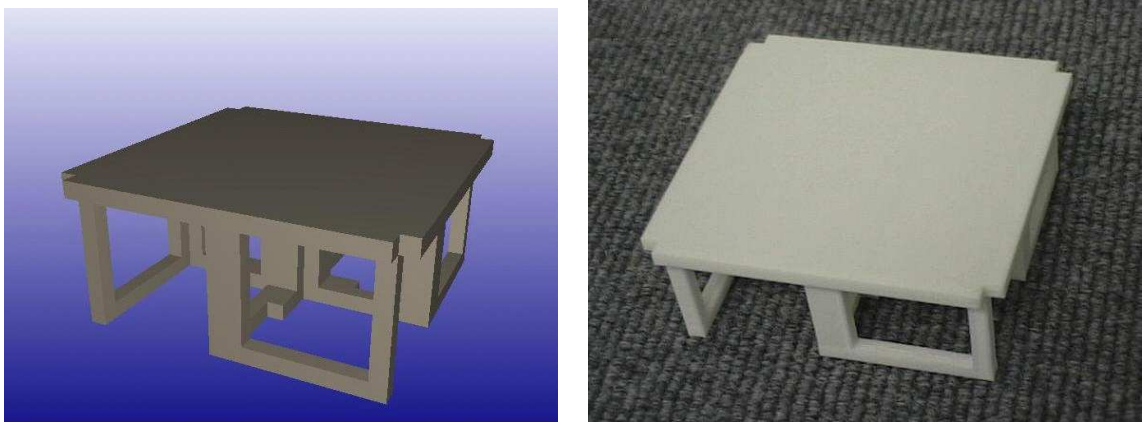
In this section it was shown that better neural-network controlled robots were evolved using the generative representation than with the non-generative representation. Taking the average over ten trials of the best individual found with each representations gives a fitness of 157 with the non-generative representation and 2609 with the generative representation. Also, with the generative representation, symbols were reused 11.5 times, on average, in translating from the genotype to the assembly procedure.

5.5 From Design to Implementation

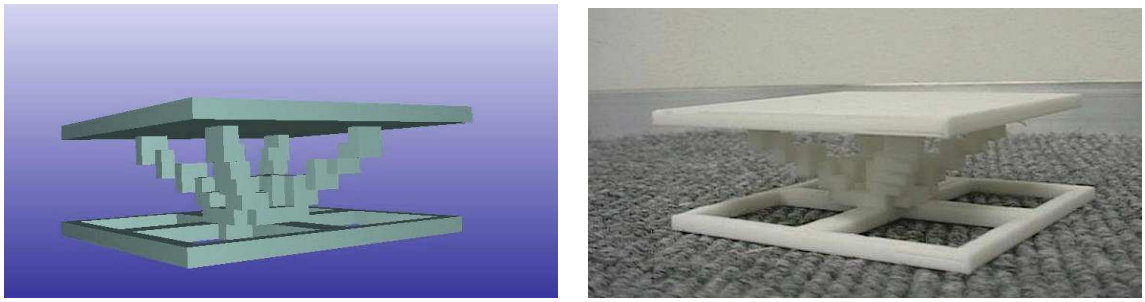
Previous work has shown the successful transfer from design to reality of static objects [48] and actuated robots [88]. Similarly, designs produced with *GENRE* have also been successfully transferred to the real world. Automated manufacture of evolved table designs was achieved by use of rapid-prototyping equipment. Evolved designs were saved as an stl file, a format for describing three-dimensional structures, and this file was sent to a rapid prototyping machine for three-dimensional printing, figure 5.26. Genobots were initially constructed using the basic parts shown in figure 4.4 and then assembled by hand, figure 5.27.a and b. More recently, genobots are built using a rapid prototyping machine in a method similar to Lipson and Pollack's GOLEM project [88], figure 5.27.c. Constructed genobots behaved in reality similar to how they performed in simulation.

5.6 Summary of Results

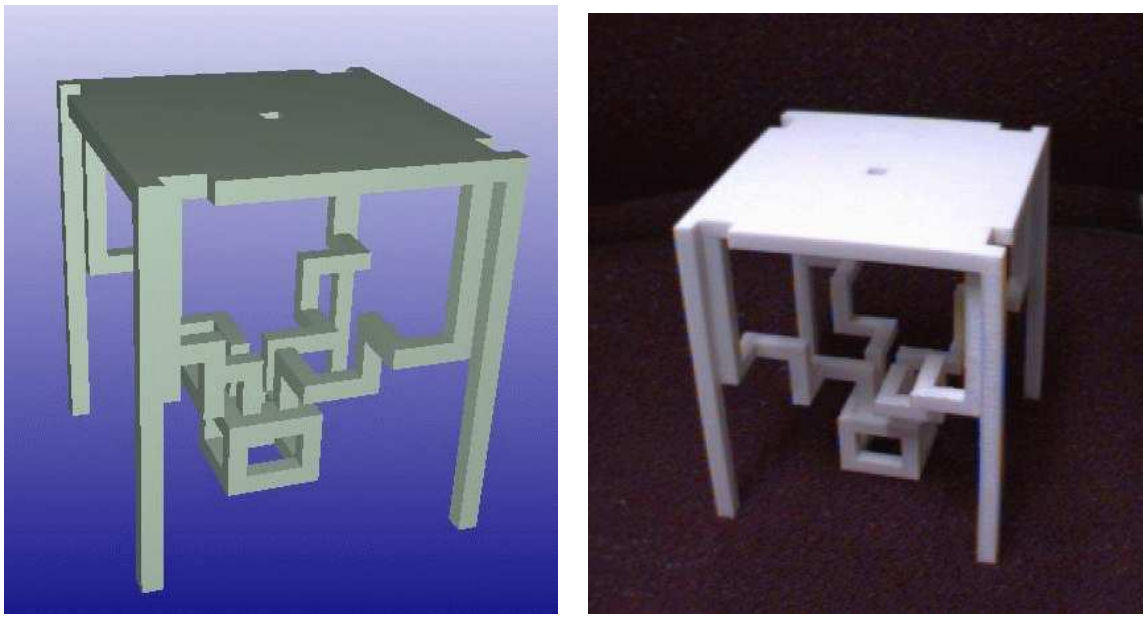
This chapter presented the results of evolving designs on four different substrates with both a non-generative and a generative representation. On all four design problems the evolutionary design system achieved better designs with the generative representation than with the non-generative representation. These results supported the argument of chapter 1 that on complex design problems the ability to reuse parts of an encoded design would improve the performance of a search algorithm. With the generative representation, average



(a1) (a2)

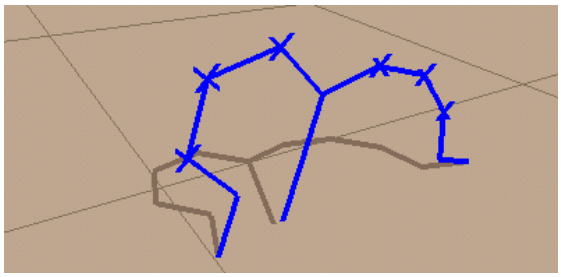


(b1) (b2)

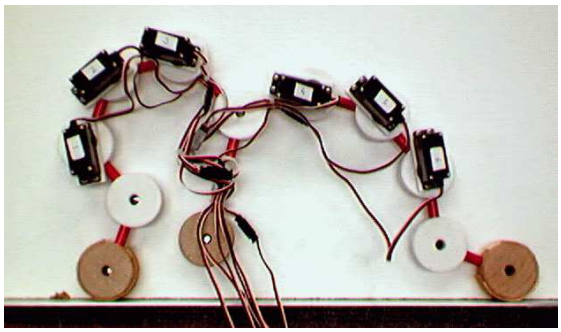


(c1) (c2)

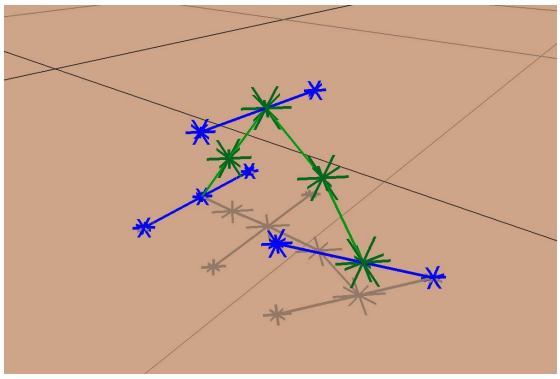
Figure 5.26: Evolved tables shown both in simulation (left) and reality (right).



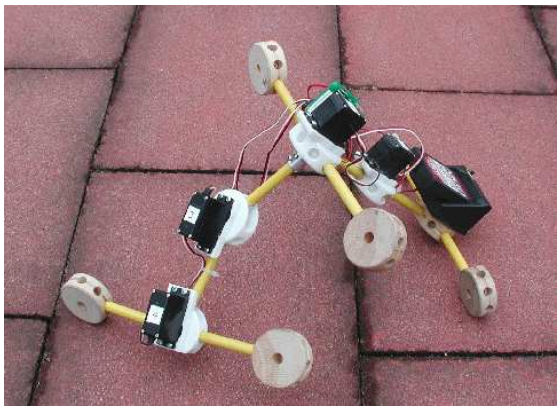
(a1)



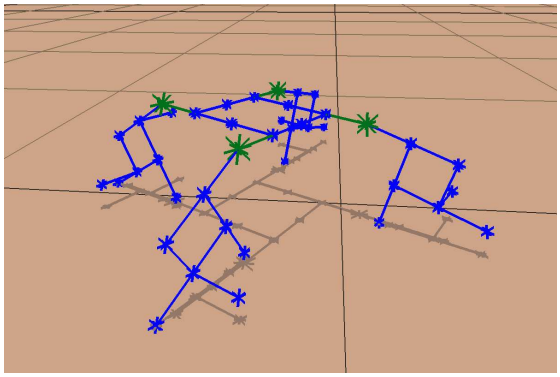
(a2)



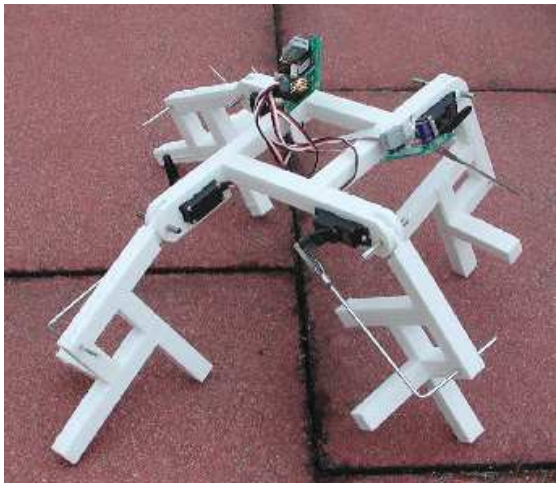
(b1)



(b2)



(c1)



(c2)

Figure 5.27: Evolved robots shown both in simulation (left) and reality (right).

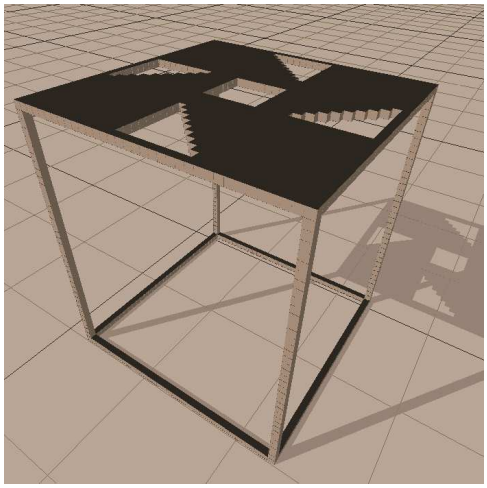
reuse of elements of encoded designs was 16.1 on table designs, 1.6 on parity networks, 12.4 on oscillator controlled robots and 11.5 on neural network controlled robots. Finally, real-world implementations of designs produced with this system were shown.

Chapter 6

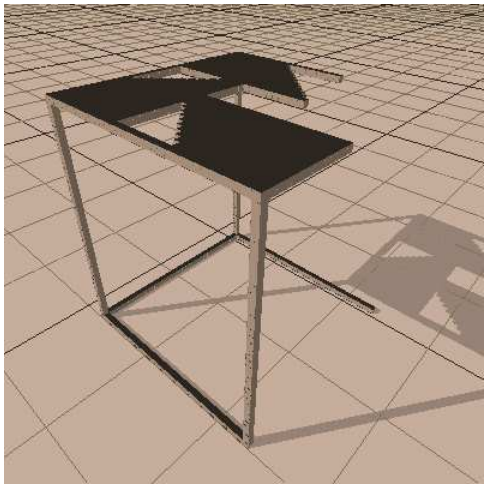
Discussion

The central claim of this dissertation is that using generative representations improves the evolvability of evolved designs and also increases the size of the design space explored with the evolutionary algorithm. The two advantages of superior evolvability of designs encoded with a generative representations and improved ability to explore large design spaces are related to each other. If designs created with the automated design system are more evolvable in the types of variation that are successful, this enables the search algorithm to explore a wider range of designs. This can be intuitively understood by looking at some examples of designs evolved with a generative representation.

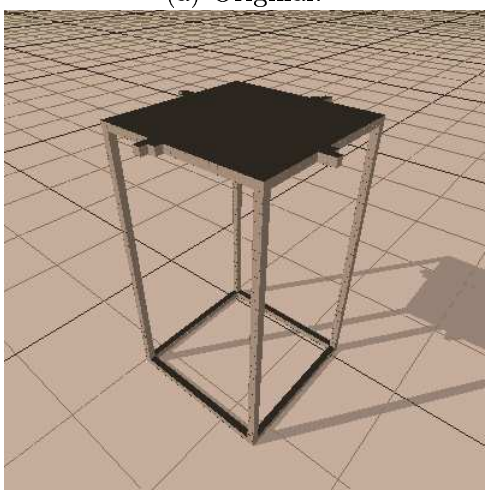
An advantage of the design encodings that are found with a generative representation is in the changes that can be made to this encoding that can not be made to designs encoded with a non-generative representation. Using the table from appendix A, figure 6.1 contains examples of different tables that can be produced with a single change to the encoded design. One change to the encoding of the table in figure 6.1.a can produce a table with: three legs instead of four by changing the parameter in the block replication command in $P0$, (b); a narrower frame, by changing symbols in either productions $P17$ or $P18$, (c); shorter legs, by changing symbols in the first successor of $P8$, (d); a surface with less voxels, by changing symbols in $P6$ or $P8$, (e); or more voxels on the surface, again by changing symbols in $P6$ or $P8$, (f). The images in figure 6.2 are another example which shows the benefits of reuse



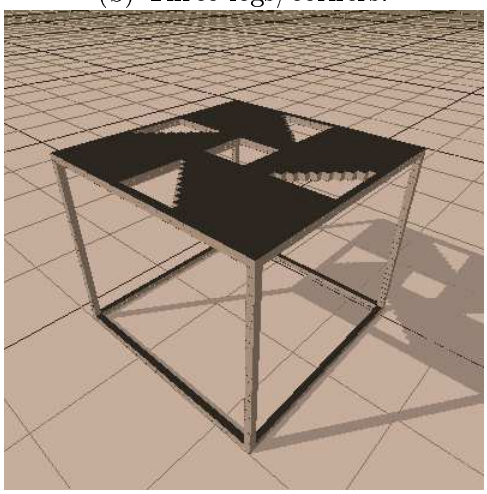
(a) Original.



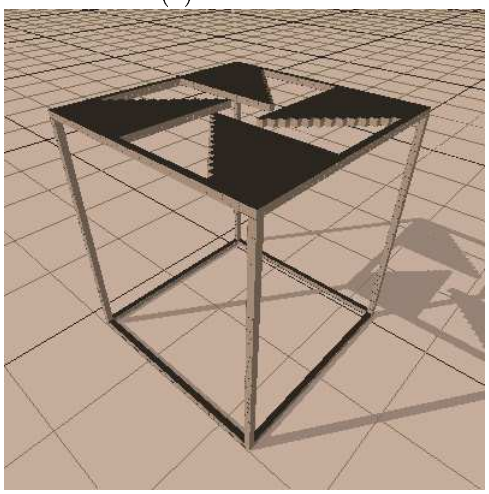
(b) Three legs/corners.



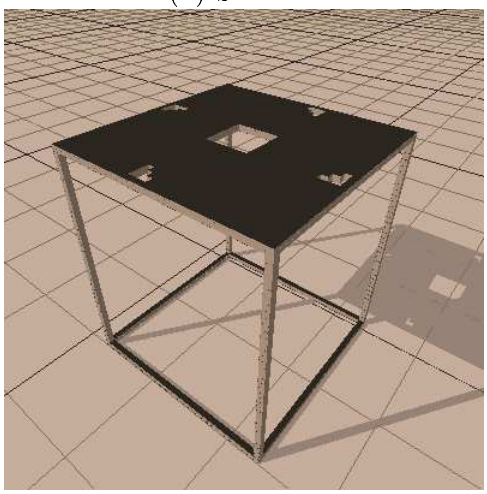
(c) Narrower.



(d) Shorter.

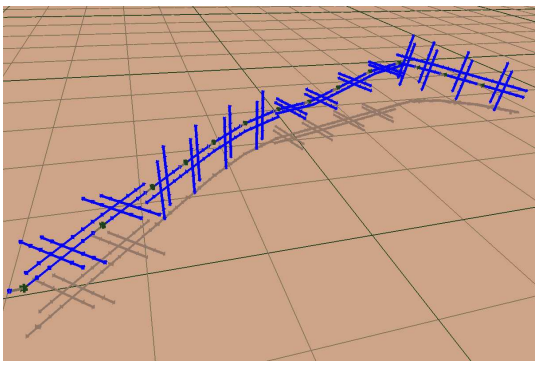


(e) Fewer surface voxels.

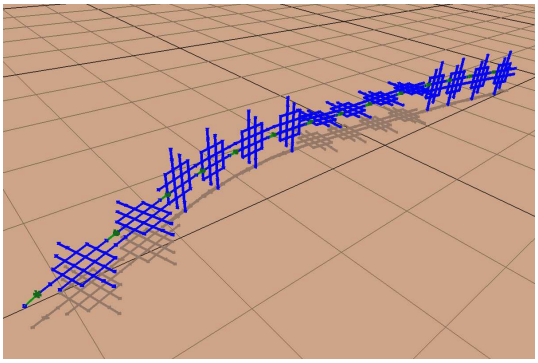


(f) More surface voxels.

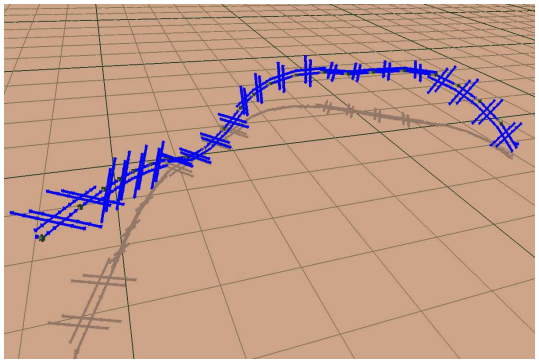
Figure 6.1: Mutations of a table.



(a)



(b)

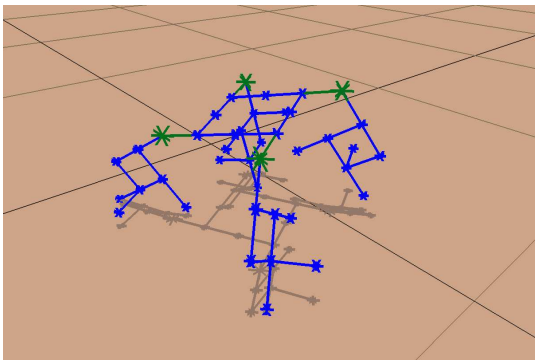


(c)

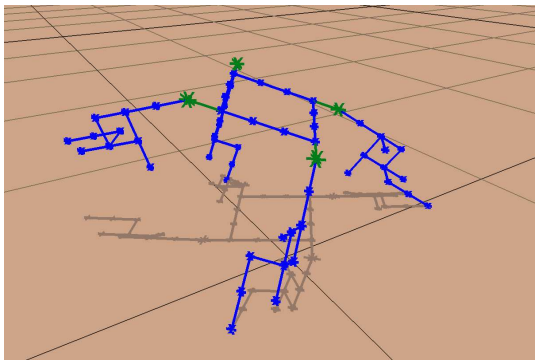
Figure 6.2: Mutations of a genobot: (a), the genobot from figure 5.22.b; (b), a change to a low-level component of parts results in all occurrences of this part to have the change; (c), a single change to the genotype changes the number of high-level components in the genobot from four to six.

through variations applied to the individual in figure 5.23.d. Changing the genotype to add rods to an assembly of parts results in the change to all occurrences of that part in the design, figure 6.2.b, and a single change to the genotype can cause the addition/subtraction of a large number of parts, figure 6.2.c. With a non-generative representation, these changes would require the simultaneous changes of multiple symbols in the encoding. Some of these changes must be done simultaneously for the resulting design to be viable – changing the height of only one leg of the table can result in a significant loss of fitness – and so these changes are not evolvable with a non-generative representation. Others, such as the number of voxels on the surface, are viable with a series of single-voxel changes. Yet, in the general case this could result in a significantly slower search speed in comparison with a single change to a table encoded with a generative representation.

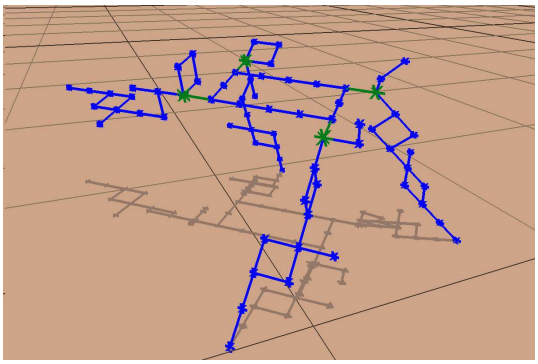
That the evolutionary design system is taking advantage of the ability to make coordinated changes with a generative representation is demonstrated by individuals taken from different generations of the evolutionary process. The sequence of images in figure 6.3, which are of the best individual in the population taken from different generations, show two changes occurring. First, the rectangle that forms the body of the genobot goes from two-by-two (figure 6.3.a), to three-by-three (figure 6.3.b), to two-by-four (figure 6.3.c), before settling on two-by-three (figures 6.3.d-f). These changes are possible with a single change on a generative representation but cannot be done with a single change on a non-generative representation. The second change is the evolution of the genobot's legs. That all four legs are the same in all six images strongly suggests that the same module in the encoding is being used to create them. As with the body, changing all four legs simultaneously can be done easily with the generative representation by changing the one module that constructs them, but would require simultaneously making the same change to all four occurrences of the leg assembly procedure in the non-generative representation. Figure 6.4 contains a sequence of images from the evolution of a neural-network controlled genobot. Being able to repeat the main body allowed search to move from the design in figure 6.4.b to figure 6.4.c to the design in figure 6.4.d. These changes to a design can be done with a few changes to the



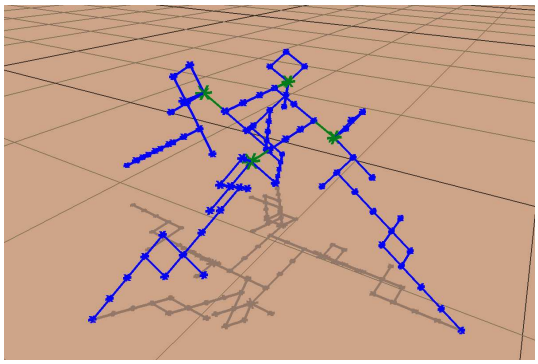
(a) Fitness: 348.



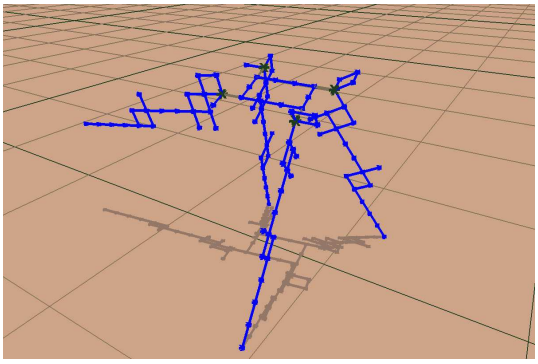
(b) Fitness: 780.



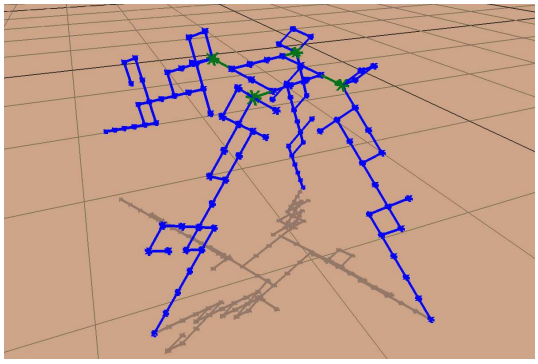
(c) Fitness: 1168.



(d) Fitness: 1450.

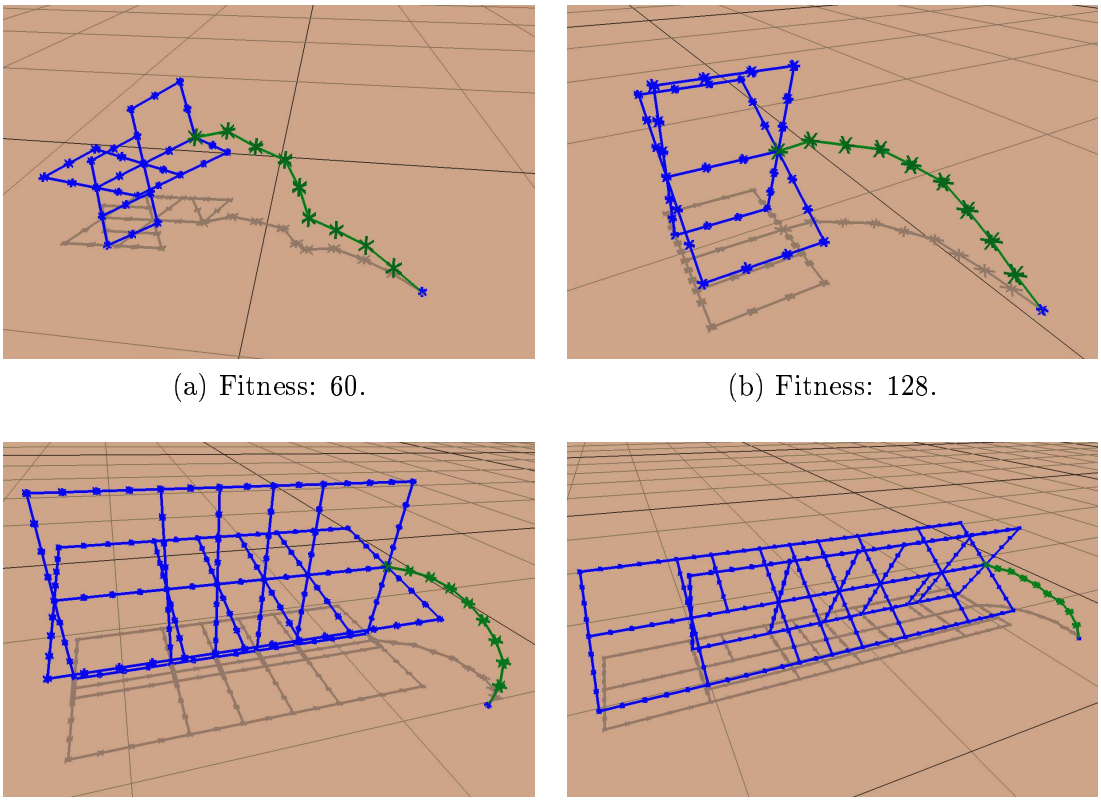


(e) Fitness: 2168.



(f) Fitness: 2192.

Figure 6.3: Evolution of a four-legged walking genobot.



(a) Fitness: 60.

(b) Fitness: 128.

(c) Fitness: 256.

(d) Fitness: 258.

Figure 6.4: Evolution of a rolling genobot.

generative representation, yet would require the coordinated changes of a greater number of symbols with a non-generative representation.

The above examples illustrate how reuse in design encodings can result in more evolvable design encodings that improve the ability of a search algorithm to explore large design spaces. The following two sections compare both the evolvability of designs evolved with the generative representation against those evolved with the non-generative representation and also compare the size of the search spaces explored by the evolutionary algorithm with the two different representations.

6.1 Evolvability

Reusing elements of an encoded design to reuse parts in the actual design with a generative representation makes certain types of design changes easier than with a non-generative representation. With a non-generative representation, as the number of dependencies in a design increase the likelihood of the right change happening simultaneously to all the dependent parts of a design becomes increasingly unlikely. Changing the height of a table, for example, requires changing each occurrence of a table leg in the genotype. In contrast, if a generative representation uses a single module to construct a table leg, changing the height of the table requires only one change in the leg-building module. While recombination can duplicate assemblies of parts in a non-generative representation, a later application of variation would only change one instance of this assembly. By capturing dependencies through shared parameters and assemblies of encoded elements, coordinated changes in the expressed design can be realized through a single change in the generative design representation. This section contains examples from the different design domains which show that the generative representation captured design dependencies in its structure, thereby improving the evolvability of an encoded design.

To determine if design encodings evolved with the generative representation incorporated useful bias of the problem into their structure, the evolvability of encodings evolved with

the two representations is compared. Measuring the evolvability of a representation depends on the meaning of evolvability. One meaning of evolvability is the ability of a genotype to produce offspring of higher fitness [7] [134]. Using this meaning, evolvability can be measured by plotting the number of offspring that fall at a given fitness differential from their parents. Another meaning of evolvability is the degree to which changes in the genotype can result in new, and useful, phenotypes [29], or the ability for the representation to facilitate large changes in the phenotype. This second meaning of evolvability can be measured by plotting the phenotypic distance against the fitness differential for offspring.

To measure evolvability, two metrics were used to compare the difference between a parent and child's assembly procedures. One metric used is the *edit distance* between two strings [112]. Because calculating the edit distance is computationally expensive this metric is used for only a single experiment in which evolved assembly procedures averaged less than a thousand commands. A second metric, called *command difference*, is used as a less computationally expensive estimate of distance between two assembly procedures. If the assembly procedures are the same length, the command difference returns the number of symbols that are different for each index (analogous to the Hamming distance between binary strings). The command difference between strings of different lengths consists of counting the number of occurrences of a symbol for each symbol type and then summing the absolute difference between these values. For example, the command difference between `abcabc` and `cbacba` is four and the command difference between `abcabc` and `cbacb` is one.

The rest of this section compares the evolvability of designs produced with the generative representation against those produced with the non-generative representation. Evolvability of the two representations, both the ability to produce offspring of higher fitness and the ability to make changes to a design, are examined on all four design substrates. In addition, with the 7-parity problem, plots using command difference and edit distance are compared and it is shown that using command difference is a near approximation of edit distance.

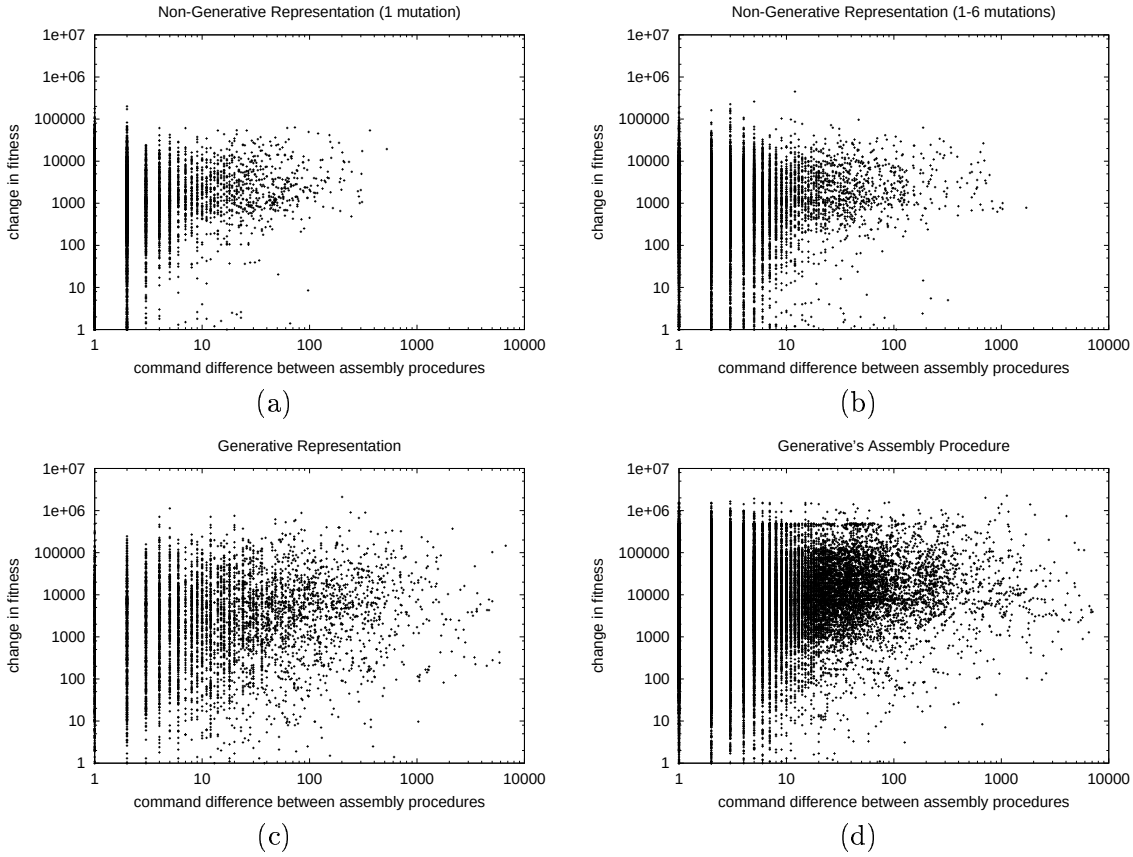


Figure 6.5: Tables: plot of amount of change in genotype from parent to child versus change in fitness (one out of every four data points) for cases with positive change.

6.1.1 Evolvability of Table Designs

To determine if the generative representation was incorporating useful bias of the design problem into its structure, the fitness difference between a parent and child is plotted along with the command difference between their assembly procedures. The graphs in figure 6.5 are scatter plots of the command difference between parent and child's assembly plans against their change in fitness. Because each of these data files contain over fourteen million data points, and most changes result in a loss of fitness, these graphs plot only one out of every four points and only those points in which the change in fitness is greater than, or equal to, one. These graphs show that large changes in the assembly procedure were more likely to be successful with the generative representation than with the non-generative representation, and improvements tended to have a larger increase in fitness with the generative representation.

Next, the success rate of the mutation operator on the two different representations is compared to determine if the generative representation produced design encodings that were more conducive to evolution. The graphs in figure 6.6 plot the number of offspring that fall at a given fitness differential from the parent design. These graphs show that the overwhelming majority of mutations to a design produced little or no change in fitness. While most of the remaining mutations produced a negative change in fitness with both representations, there are more positive changes to fitness with the generative representation, especially large positive changes. A plot of the success rate under mutation (a child has higher fitness than its parent) is shown in the graphs in figure 6.7. For changes of less than 400 symbols, the success rate was higher with the non-generative representation than with the generative representation. A possible explanation is that with the non-generative representation it is easier to make small changes that add voxels to the table-top than with the generative representation. Changes of more then 400 symbols tended to be more successful on the generative representation, with the non-generative representation having no successful mutations of more then 1000 symbols whereas 7425 was the largest command difference for which an improvement was made with the generative representation. These two

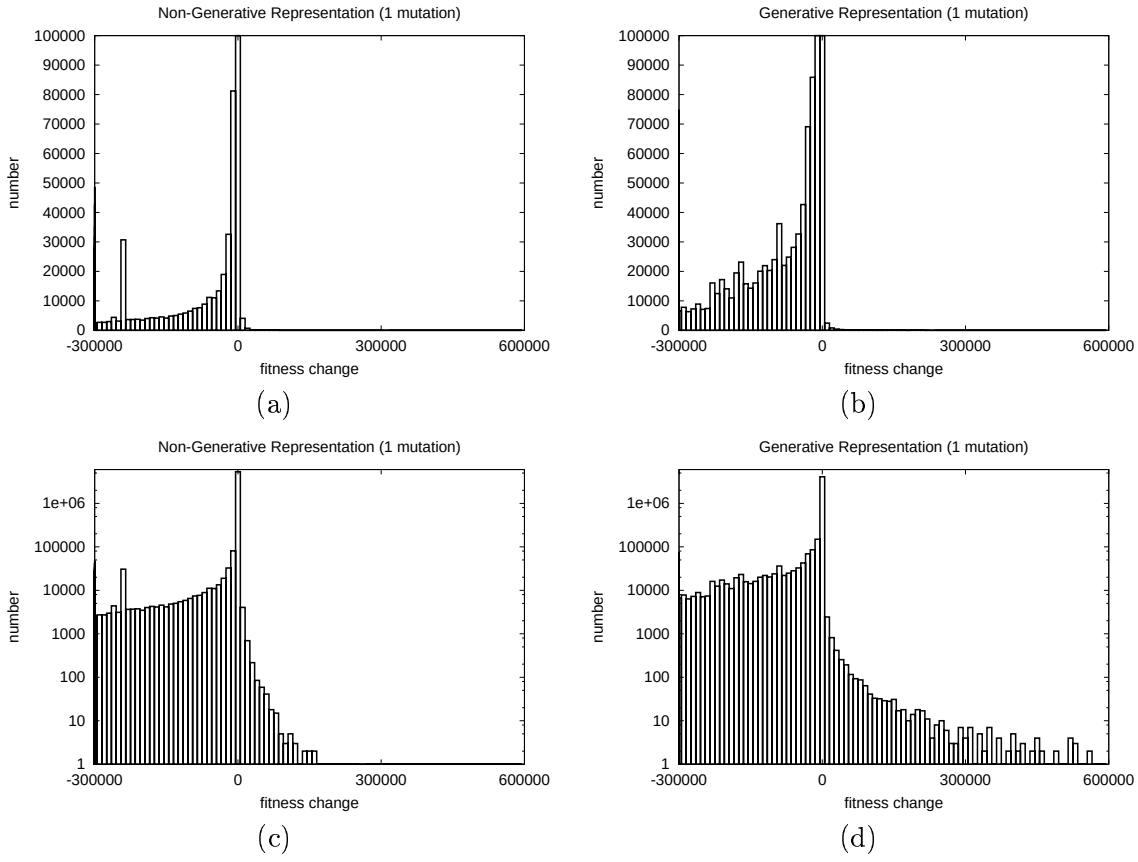


Figure 6.6: Table designs: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d).

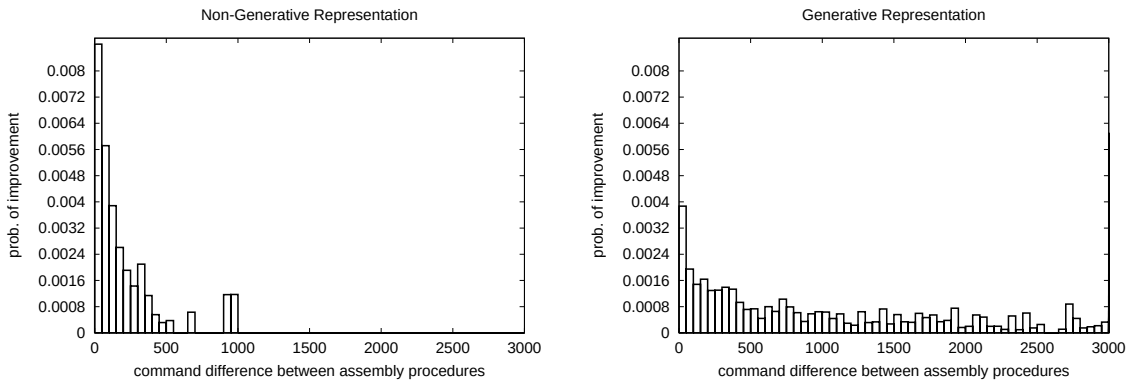


Figure 6.7: Tables: probability of improvement (child is more fit than parent) comparison between non-generative and generative representations, for ranges 1-50, 51-100, 101-150, ... 1951-2000.

sets of graphs show that encodings with the generative representation were more evolvable than the non-generative representation both in that positive changes to fitness are more likely and large changes to the design were more likely to result in a higher fitness.

6.1.2 Evolvability of Parity Networks

Graphs plotting the change in fitness against assembly procedure command-difference/edit-distance between a parent and its child are shown in figure 6.8. These graphs show that mutations on designs encoded with the non-generative representation produced less improvements in fitness than did mutations on designs encoded with the generative representation. This provides evidence that over the course of evolution, designs encoded with the generative representation captured useful properties of the design problem into their structure.

That evolutionary search with the generative representation produced more evolvable design encodings than with the non-generative representation is shown by the graphs in figures 6.9 and 6.10. The graphs in figure 6.9 plot the number of offspring at given fitness differential ranges from their parents. These graphs show that there were more positive changes in fitness with the generative representation for all ranges and the maximum positive fitness change that occurred with the generative representation was approximately twice

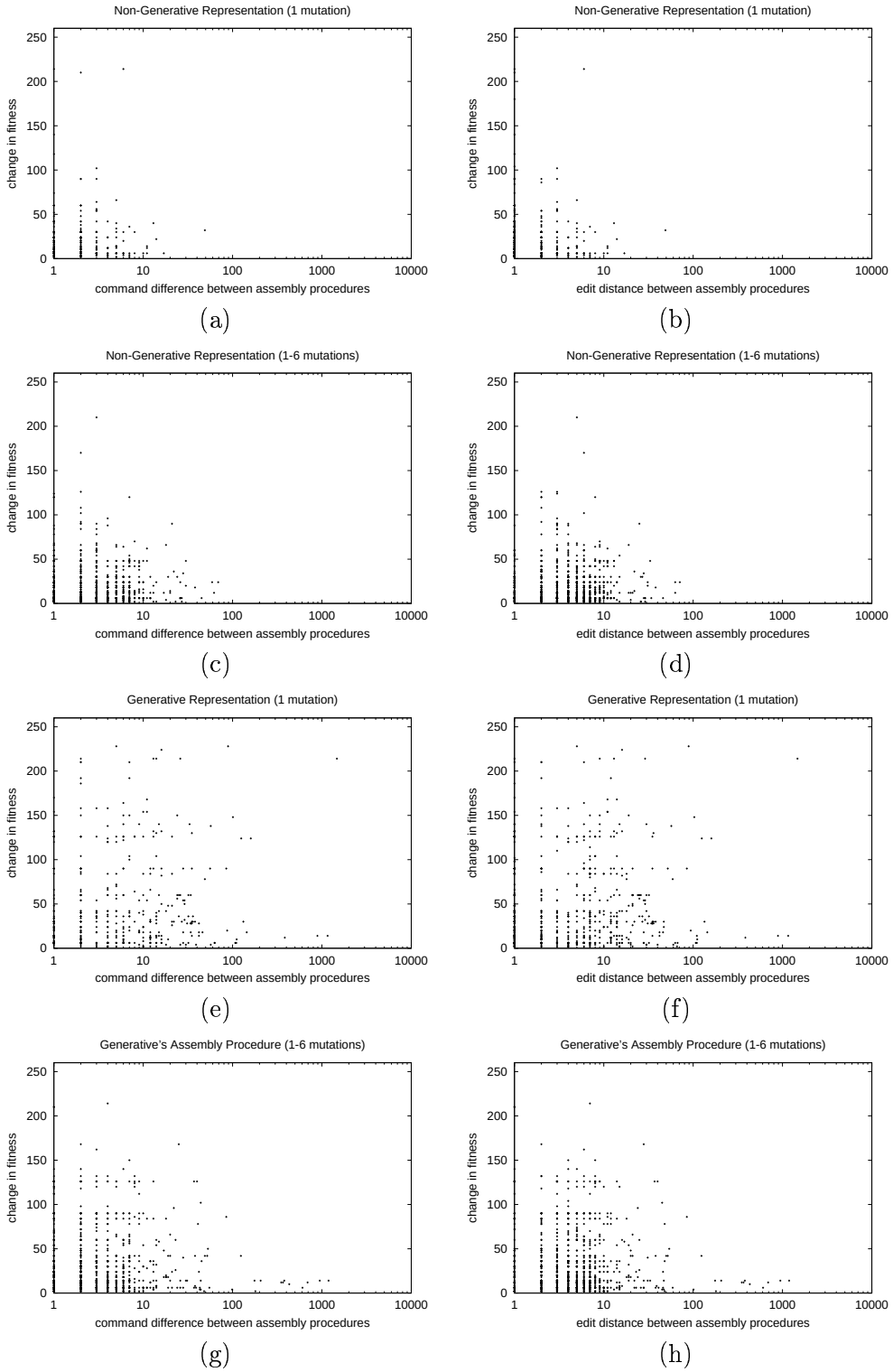


Figure 6.8: 7-Parity: plot of command-differences/edit-distances between parent and child's assembly procedures versus change in fitness (positive improvements only).

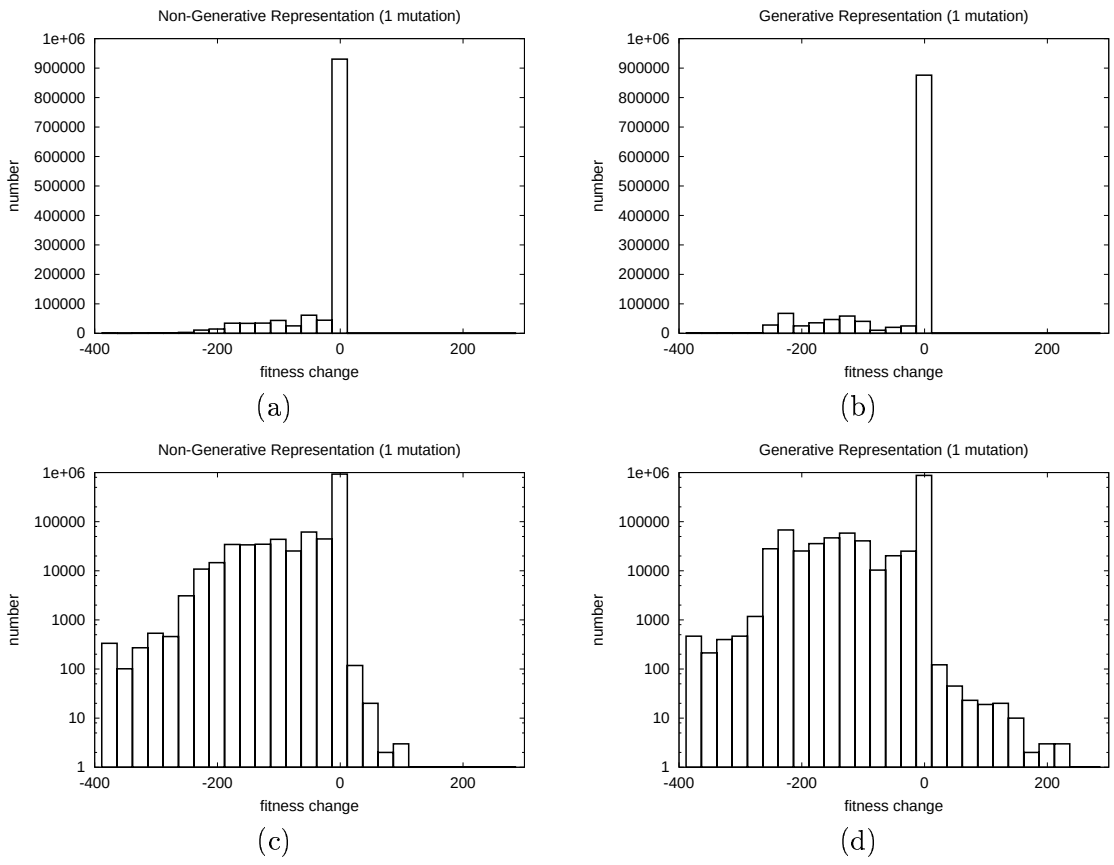


Figure 6.9: 7-Parity: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d).

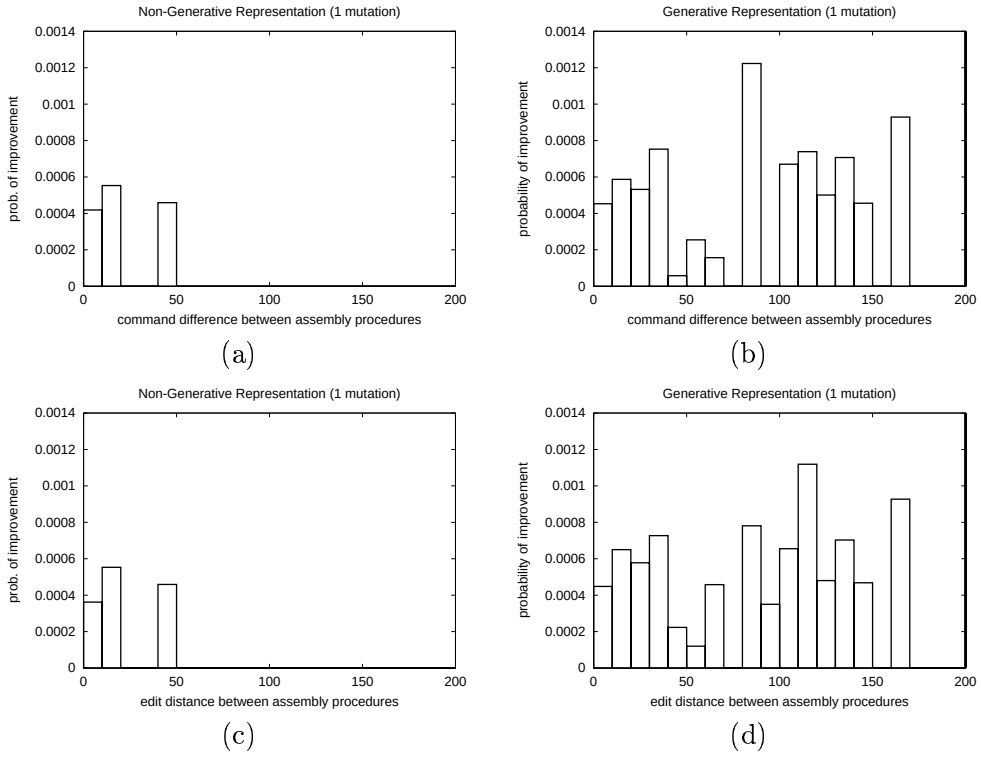


Figure 6.10: 7-Parity: probability of improvement (child is more fit than parent) comparison between non-generative and generative representations (both command difference and edit distance) for ranges 1-10, 21-30, 31-40, ... 191-200.

that with the non-generative representation. The graphs in figure 6.10 are plots of the probability of success of mutation for different ranges of command-difference/edit-distance between a parent and child's assembly procedures. These graphs show that mutations that produced large changes in an individual's assembly procedure were more likely to be successful (have a higher fitness than the parent individual) on designs encoded with a generative representation than on designs encoded with a non-generative representation. Both sets of graphs show that the generative representation was more evolvable, suggesting that over the course of evolutionary search the generative representation captured useful properties of the design problem.

Since assembly procedures evolved for this class of design problems averaged less than 250 commands, edit distance was used to compare a parent and child's assembly procedures in addition to the command difference used on all design substrates. Comparing graphs of the same individuals using the different metrics allows for a comparison of the metrics themselves. The graphs in figures 6.8 and 6.10 have similar characteristics suggesting that plots using edit distance would be similar to those using command difference on the other design problems.

6.1.3 Evolvability of Oscillator-controlled Robot Designs

In chapter 1 it was argued that a generative representation would be more conducive to large variations on the design than a non-generative representation because of its ability to capture useful bias into its structure. The graphs in figure 6.11 are scatter plots of the command difference between a parent and child's assembly procedures against their change in fitness. From the graphs in figures 6.11.a and b it can be seen that mutation was not generally beneficial on the non-generative representation, and only produced designs with higher fitness when small changes were made. Comparing the plots in figures 6.11.c and d shows that mutation was more effective on the generative representation itself than on the assembly procedure produced by the generative representation. This means that it is in the structure of the encoding of the generative representation that useful bias was captured and

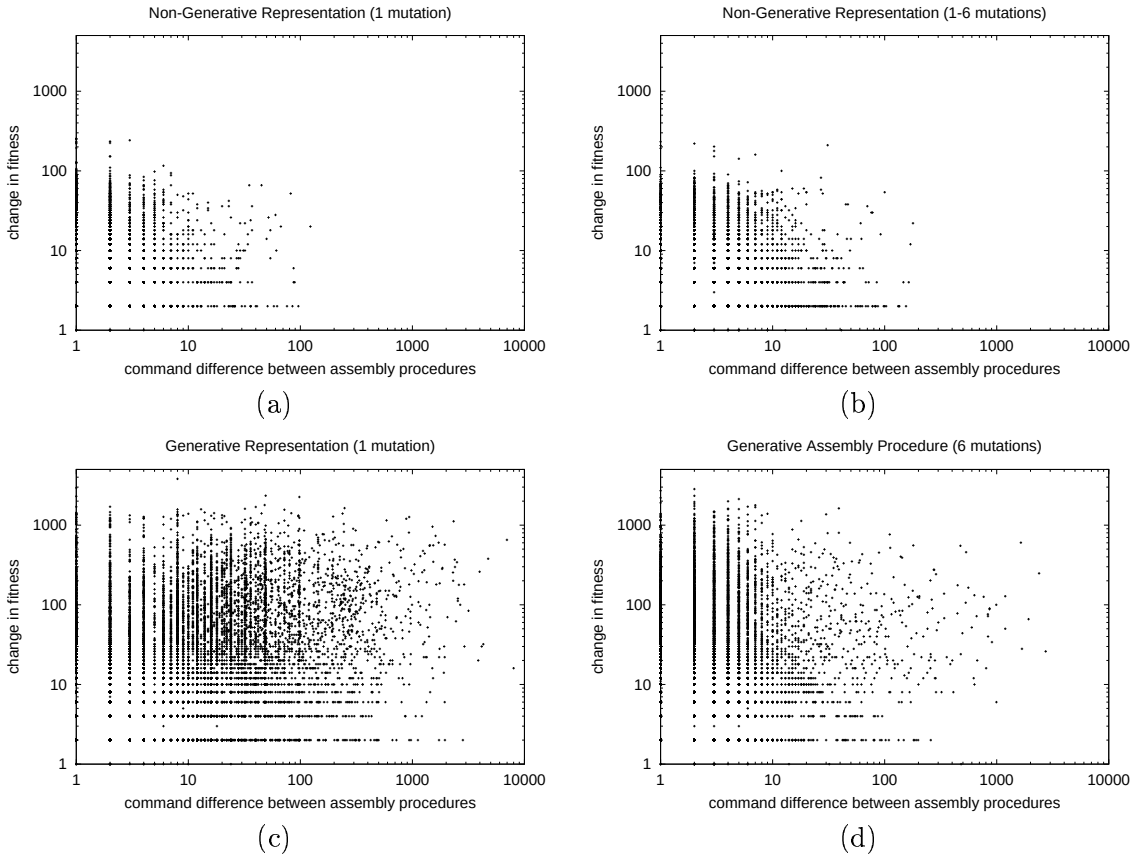


Figure 6.11: Oscillator robots: plot of amount of change in assembly procedures from parent to child versus change in fitness.

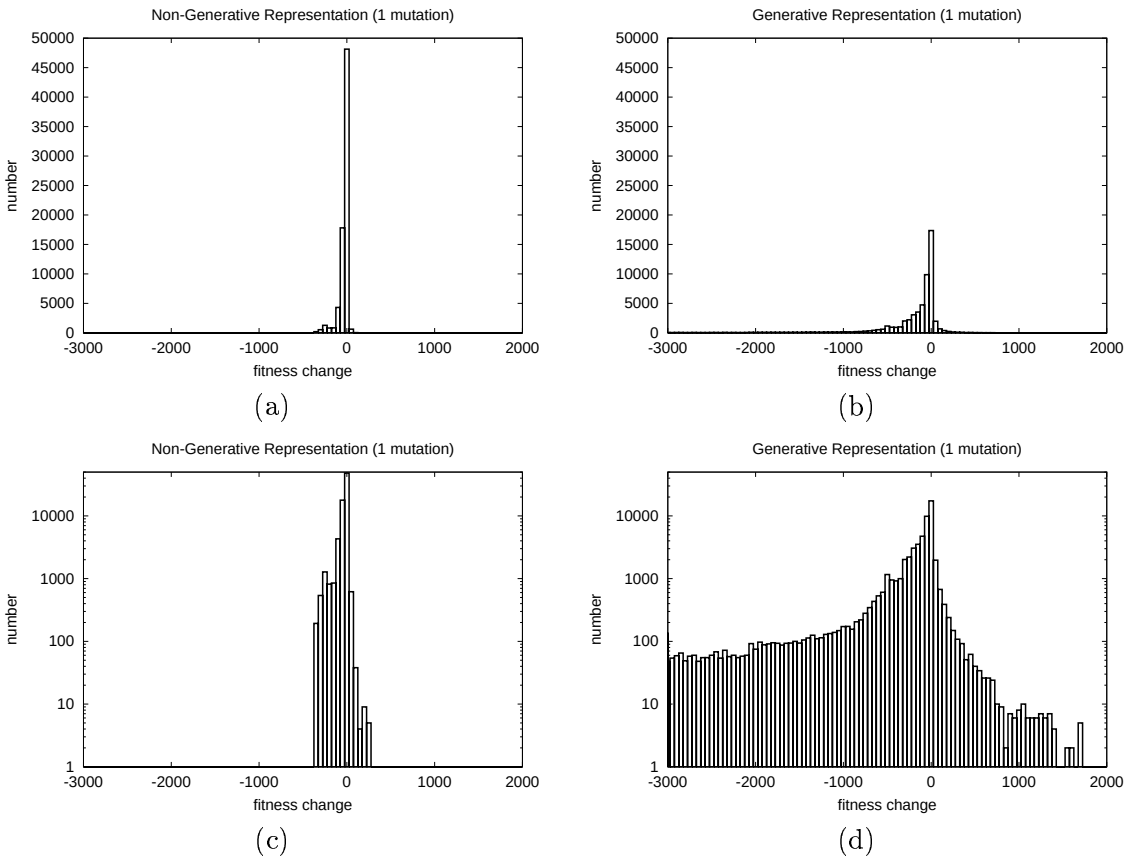


Figure 6.12: Oscillator-controlled robots: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d).

not just the in sequence of symbols it produced.

Finally, the graphs in figures 6.12 and 6.13 show the evolvability of the two representations. The graphs in figure 6.12 plot the number of offspring that had a given fitness differential from their parent. These graphs show that a larger number of offspring had higher fitness than their parent with the generative representation than with the non-generative representation. They also show that there was a greater range in fitness differentials with the generative representation than with the non-generative representation. Finally, the graphs in figure 6.13 show the rate of success of mutations (child has higher fitness than its parent) for different distances between parent and child assembly procedures. With the non-generative

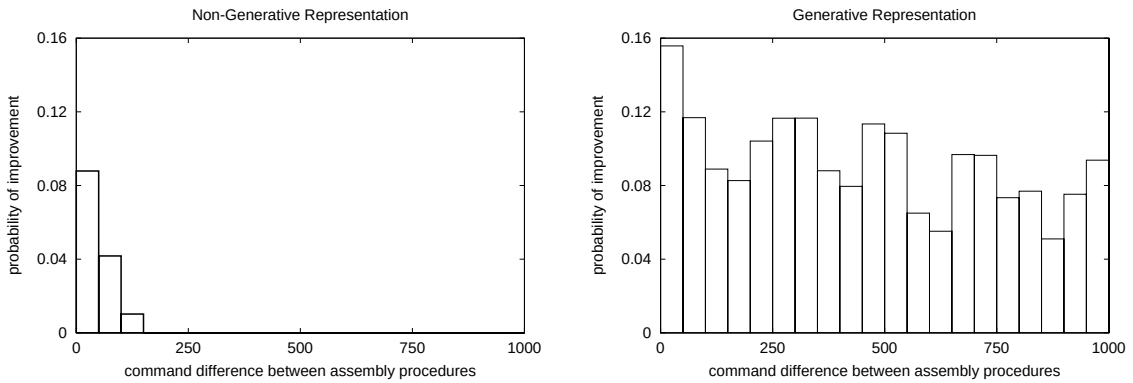


Figure 6.13: Oscillator-controlled robots: probability of improvement (child is more fit than parent) comparison between the non-generative and generative representations, for ranges 1-50, 51-100, 101-150, ... 951-1000.

representation, the success rate quickly fell to 0 for changes of more than 150 symbols in the assembly procedure. With the generative representation, the rate of success also decreased as the distance between a parent and child assembly procedures increased, but this stayed above zero well past a difference of 1000 symbols (at which point the rate of success was 8%). These two sets of graphs both show that the generative representation had more evolvable individuals than the non-generative representation.

6.1.4 Evolvability of Neural-network-controlled Robot Designs

The second part of our argument for a generative representation is that, through evolution, design dependencies become 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 plot it against the difference between parent and child's assembly procedures. For the non-generative 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

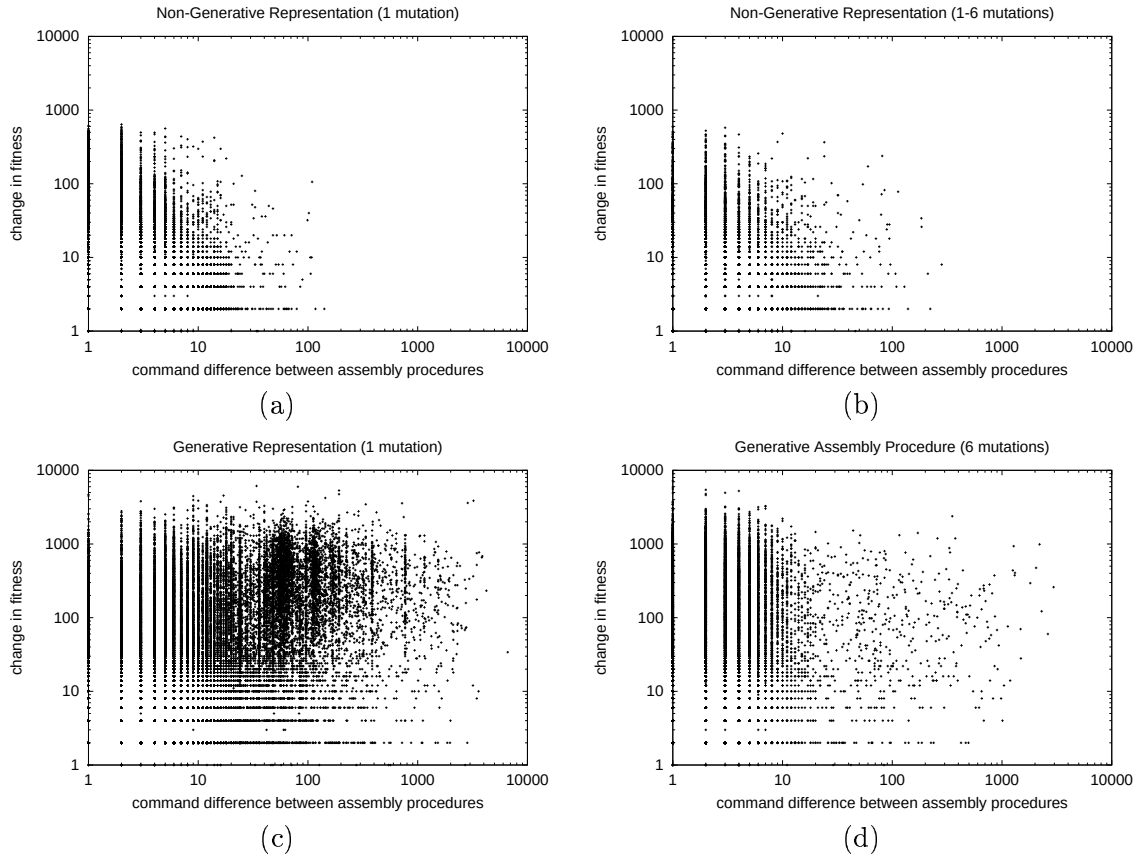


Figure 6.14: Plot of amount of change in genotype from parent to child against change in fitness.

is the number of locations for which the parent and child have different symbols. When strings are different lengths, the number of occurrences of each symbol is counted and the command difference is the sum of the differences between these values.

Figure 6.14 shows four different change-in-fitness against command-difference plots. The first graph, figure 6.14.*a*, is a plot of change-in-fitness against command difference for a single mutation operator applied to a non-generative 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 non-generative representation with 1-6 mutations (chosen with uniform probability) applied. From the graphs it can be seen that mutations on the non-generative representation were usually only successful 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 6.14.c is a plot 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 non-generative representation. This graph also shows offspring were more likely to have higher fitness than their parents with the generative representation than with the non-generative 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 one to six mutations to the assembly procedure produced by the generative representation. The plot of figure 6.14.d shows that variation on the generative representation's assembly procedure was not as successful as on 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.

Next the evolvability of the two representations are compared. The graphs in figure 6.15 are plots of the distribution of fitness differentials between a parent and its child for both representations. Comparing the two distributions shows that more offspring had a higher

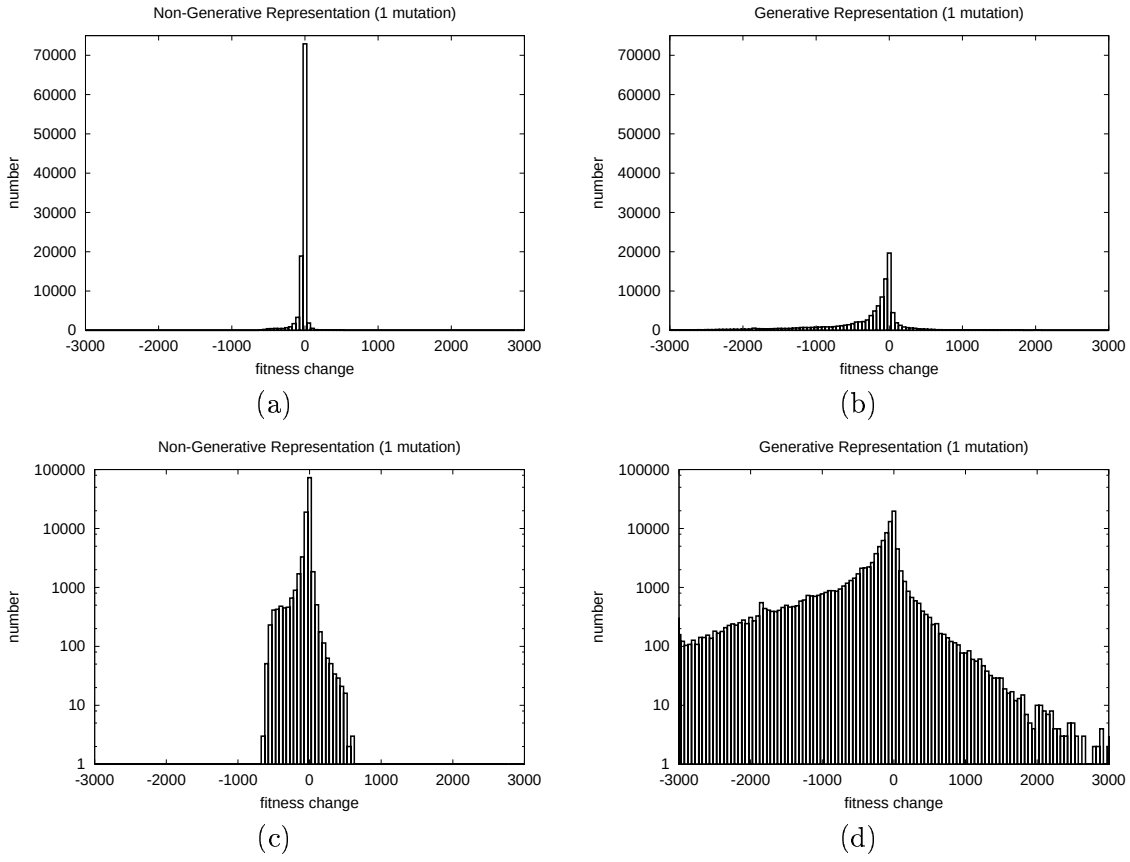


Figure 6.15: Neural network controlled robots: regular and log graphs of the number of offspring that had a given fitness differential from their parent with: the non-generative representation, (a) and (c); and the generative representation, (b) and (d).

fitness than their parent with the generative representation than with the non-generative representation. These graphs also show that the distribution of offspring extended out beyond a positive change of 3000 in fitness with generative representation but did not extend beyond 700 with the non-generative representation. This shows that evolution with the generative representation create more evolvable design encodings than with the non-generative representation for this design problem.

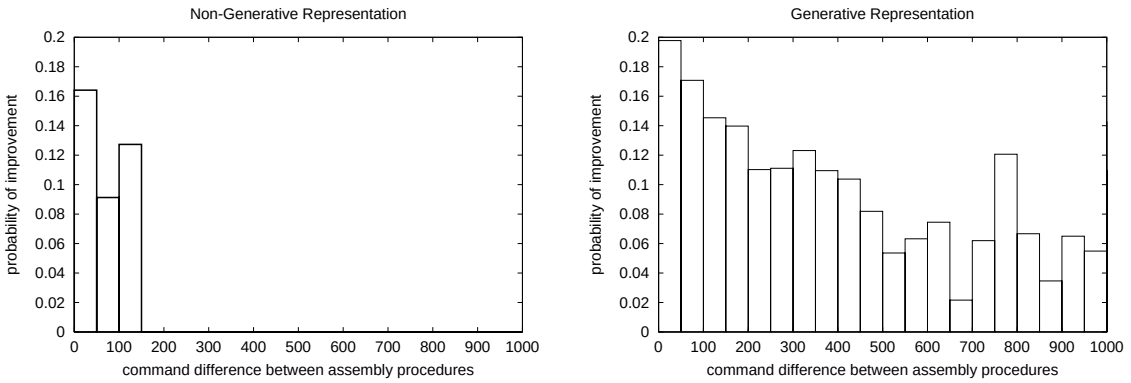


Figure 6.16: Neural network controlled robots: probability of improvement (child is more fit than parent) comparison between different representations, for ranges 1-50, 51-100, 101-150, ... 951-1000.

As a way of normalizing for the higher average fitness achieved with the generative representation, we next show the success rate (a child's fitness is greater than its parent) for the mutation operator for different command differences between parent and child, figure 6.16. Again we include a comparison with 1-6 mutations applied to the non-generative representation as well as 1-6 mutations applied to the assembly procedure produced by the generative representation. With the non-generative representation, the success rate of the mutation operator quickly dropped to zero as the difference between parent and child's assembly procedures increased. In contrast, the success rate of mutation more gracefully decayed with command difference when applied to the generative representation – even when parent and child were 500 construction symbols apart the success rate of mutation 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 provides strong evidence that the generative representation captured meaningful bias of the design problem.

6.1.5 Summary

Table 6.1: Average fitness change of successful mutations.

Substrate	Non-Generative	Generative
Tables	2826	16102
7-parity networks	19	51
Oscillator-controlled robots	10	76
Network-controlled robots	19	178

Table 6.2: Average command difference of successful mutations.

Substrate	Non-Generative	Generative
Tables	3	71
7-parity networks	2.6	22.1
Oscillator-controlled robots	2	46
Network-controlled robots	3	44

In chapter 1 it was argued that on complex design problems a generative representation would capture structure of the design problem through reuse of elements of the genotype, thereby improving the evolvability of encoded designs. In this section two definitions of evolvability were given and the evolvability of designs produced with the generative representation was compared against designs produced with the non-generative representation. The first type of evolvability is defined as the ability for changes to a design to result in offspring of higher fitness. Table 6.1 summarizes the comparison between the generative and non-generative representation for this second type of evolvability and shows that the average improvement of offspring was higher with the generative representation than it was with the non-generative representation. The second type of evolvability is defined as the ability to make large changes to a design. Table 6.2 summarizes the results of measuring

the this type of evolvability for the two types of design representations and shows that the average size of a successful design change was larger with the generative representation than with the non-generative representation. The higher average increase in fitness and the larger average command difference for successful mutations both show that search with the generative representation produced more evolvable design encodings than the non-generative representation.

6.2 Searching Large Design Spaces

The second advantage of using a generative representation comes from its superior ability to explore large design spaces. The graphs in figure 6.17 are scatter plots of the number of parts against the fitness with the non-generative and generative representations for each of the design problems. These graphs show that with the generative representation, the search algorithm tested designs with a wider range of parts and fitness than with the non-generative representation. The scatter plots in figure 6.18 are plots of the size of the encoded designs against the fitness of the designs. The graphs show that search with the generative representation typically found better designs with shorter representations than did search with the non-generative representation.

Even if a representation is better able to encode designs with a large number of parts and many dependencies this leaves the problem of how to explore an exponentially larger design space. For example compare the size of search spaces used in searching design spaces of n parts. With a non-generative encoding the genotype's length is directly proportional to the number of parts in the artifact, and the search space is some exponential on this length, k^n. If it is assumed that on average each part of the genotype is used twice in constructing the phenotype (taking into account extra genotype needed to specify how to construct the phenotype) then the length of a generative encoding for a problem with n parts is $\frac{n}{2}$, which has a search space of size $k^{\frac{n}{2}}$. In fact the results of chapter 5, and presented again in figure 6.19, show that with the generative representation the degree of reuse of elements in

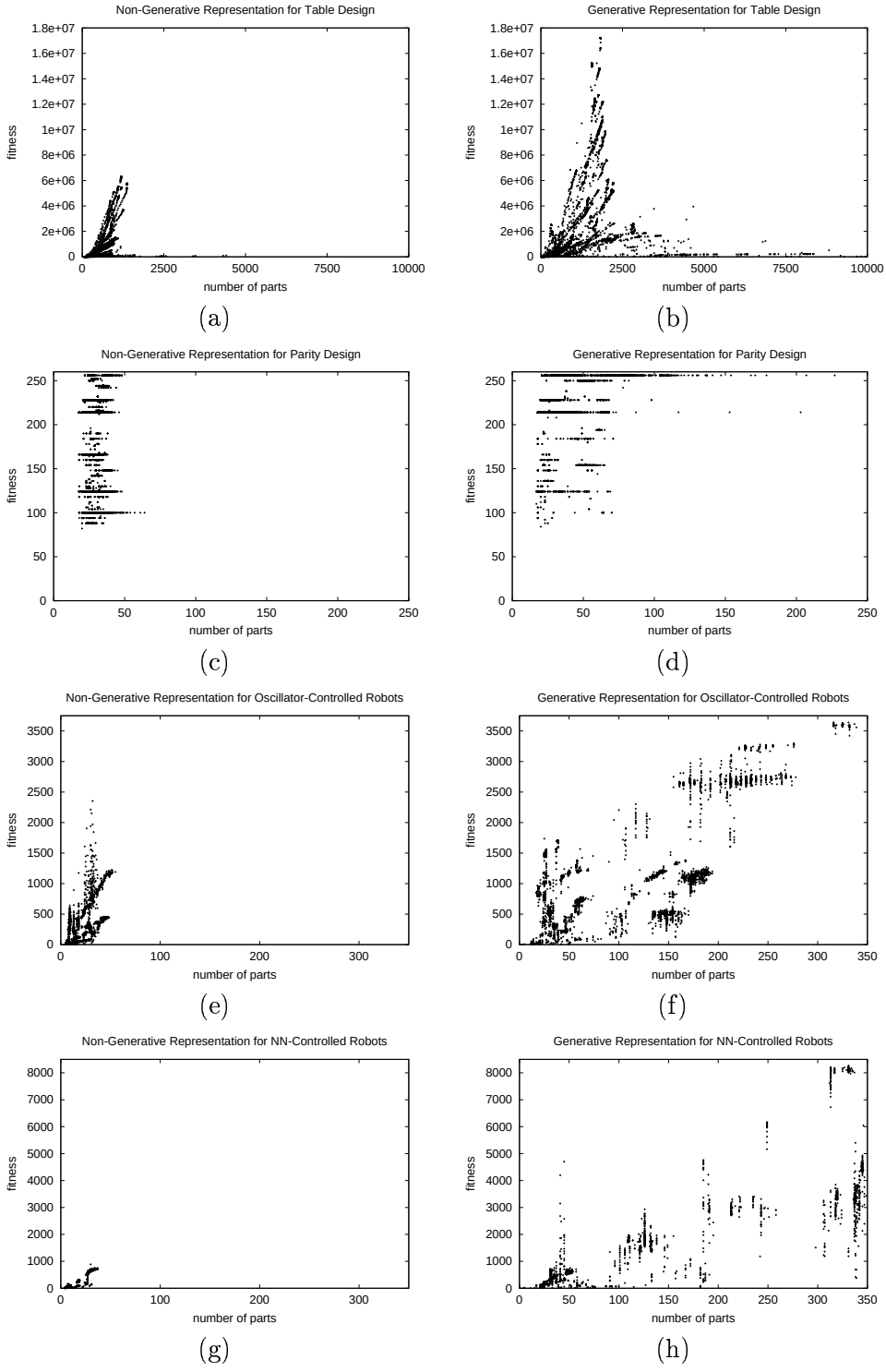


Figure 6.17: Plots of number of parts versus fitness of the best individual from each trial: (a) and (b) are for tables (plotting one out of every ten points); (c) and (d) are for parity-solving neural networks; (e) and (f) are for 3D, oscillator controlled robots; and (g) and (h) are for 3D, neural-network controlled robots.

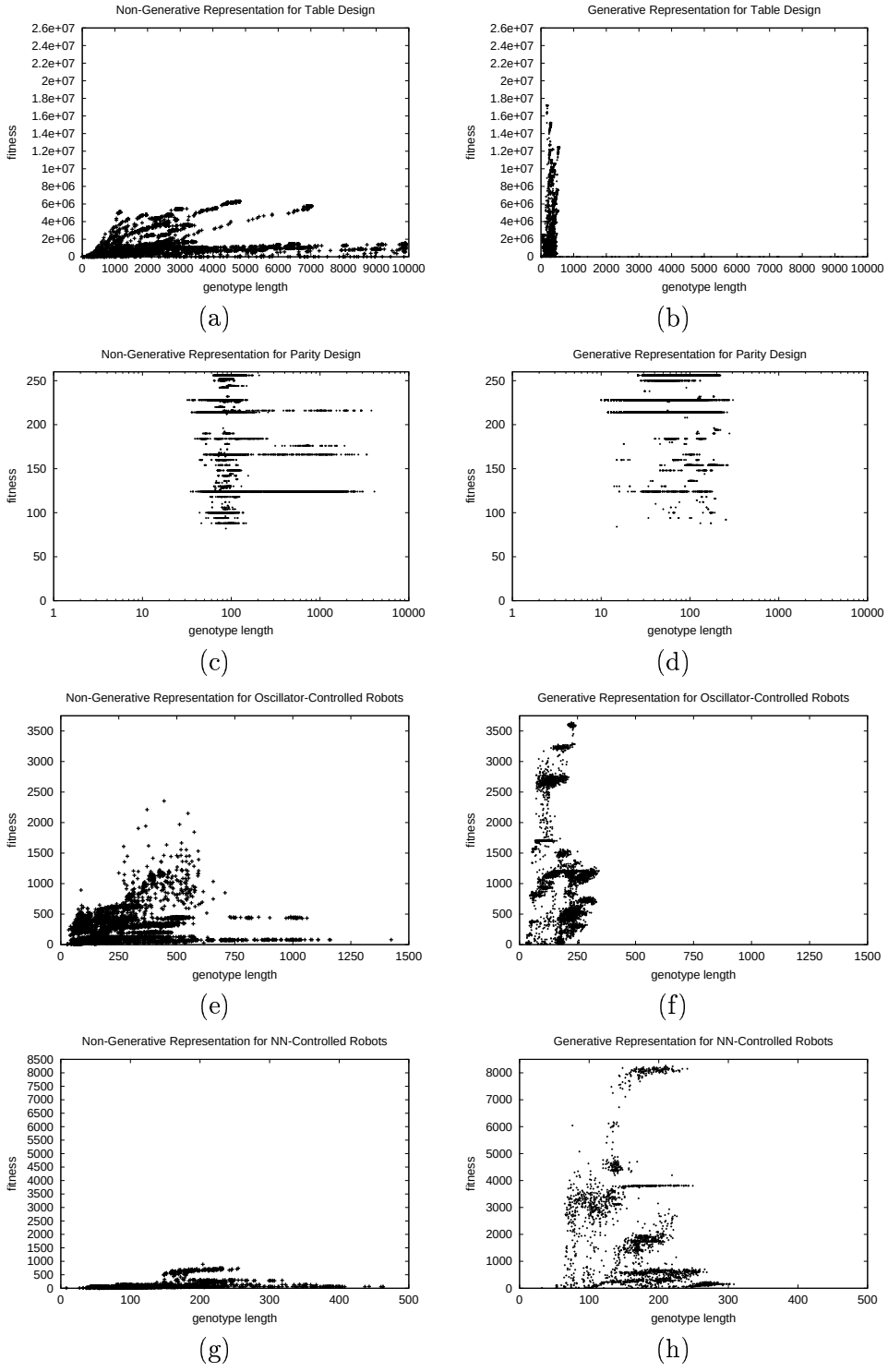


Figure 6.18: Plots of genotype length versus fitness of the best individual of each generation from each trial: (a) and (b), table designs (one out of every ten points); (c) and (d) are for parity-solving neural networks; (e) and (f), 3D, oscillator controlled robots; and (g) and (h), 3D, neural-network controlled robots.

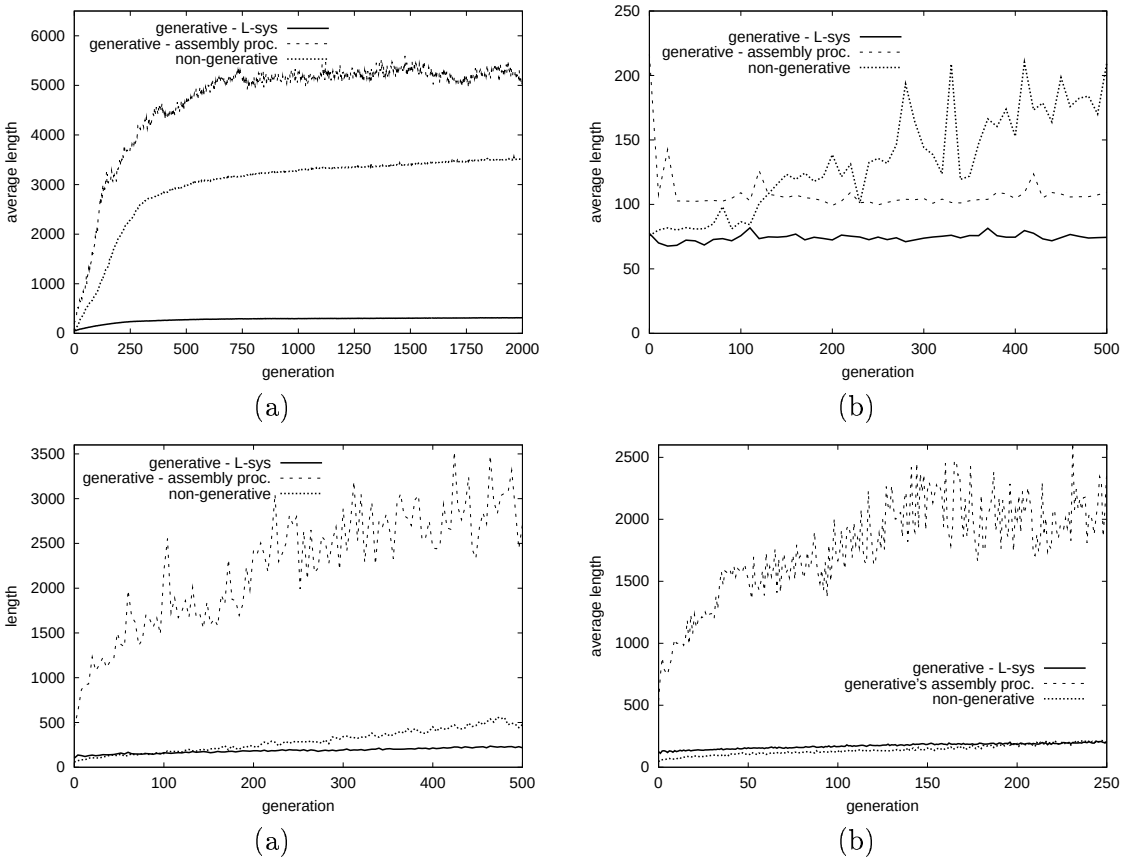


Figure 6.19: Graph of length of the genotypes and assembly procedure produced by the generative representation against generation for: (a) table designs; (b) parity networks; (c) oscillator controlled robots; and (d) neural-network controlled robots.

the genotype was sixteen for tables designs, one and a half for parity networks, twelve for oscillator controlled robots and eleven for neural-network controlled robots.

For this reduction to lead to better scaling in automated design, the inductive bias of which parts of the design space to leave out and which parts to focus on must be automated and not performed by the researcher. Since the bodies of the production rules are changed over the course of evolutionary search, it is the mapping function from encoded design to artifact that is being optimized. Thus with the generative representation, the inductive bias is captured over the course of evolution and not programmed by the researcher.

6.3 Summary

In summary, generative representations can capture design dependencies in ways more conducive to evolution than non-generative representations, which results insearch with generative representations being better able to explore large design spaces than search with a non-generative representation. The images in figures 6.1 and 6.2 show how reuse of encoded design elements resulted in the evolution of design encodings with a meaningful modularization of the design, allowing some large-scale design changes to be made easily. Examples showing the evolutionary algorithm making large-scale changes to a design are shown in the images in figures 6.3 and 6.4. In section 6.1 it was shown that through reuse generative representations captured dependencies of evolved designs, thereby allowing for more evolvable design encodings. Finally, in section 6.2 it was shown that the ability to reuse elements of the encoded design resulted in the evolutionary algorithm searching a larger area of the design space than with the non-generative representation.

Chapter 7

Conclusions

7.1 Summary

This thesis investigated representations for evolutionary design systems. First, the field of programming languages was used to identify properties of design representations as: combination, iteration, labels to compound elements, and parameterization of these labels. Using these properties generative representations were defined as those representations which allow for reuse of elements in design encodings through either iteration or abstraction. Next it was argued that, with the same basic command set, generative representations would have better scaling properties than non-generative representations through their ability to incorporate good bias of the design problem into their structure, better enabling the search of large design spaces. To support this claim an evolutionary design system, *GENRE*, and generative representation were described for evolving tables, neural networks and robots. Using *GENRE*, a generative representation was compared to a non-generative representation on four classes of design problems: tables [59]; recurrent neural networks; oscillator controlled robots [57] [61] [58]; and neural network controlled robots [60] [62]. The results of these comparisons showed:

- Better designs were evolved with the generative representation than with the non-generative representation (figures 5.1, 5.9, 5.14 and 5.20).

- Designs encoded with the generative representation had a reuse of parts of the genotype on average: sixteen times for tables designs, figure 5.8; one and a half times for parity networks, figure 5.13; twelve times for oscillator controlled robots, figure 5.19; and eleven times for neural-network controlled robots, figure 5.24.

- Variation on designs encoded with the generative representation were more evolvable than those encoded with the non-generative representation in two ways. First, changes to design encodings were more likely to improve a design on the generative representation than on the non-generative representation, and, of those changes that were improvements, these changes had a larger expected increase in fitness with the generative representation (figures 6.6, 6.9, 6.12 and 6.15). Second, large changes to the artifact (measured by distance between assembly procedures) were more likely to be successful with the generative representation (figures 6.7, 6.10, 6.13 and 6.16). In both cases successful changes to designs encoded with the generative representation had higher increases in fitness than changes on designs encoded with the non-generative representation.

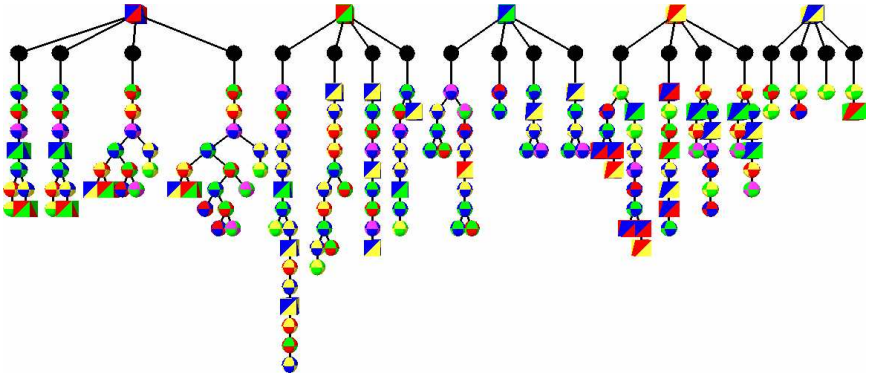
- Better evolvability of the generative representation for large changes in the design shows that the generative representation better enabled search in large design spaces than the non-generative representation.

- Reuse of genotypic elements of design encodings in conjunction with the superior evolvability of the generative representation shows that the generative representation is better capturing design dependencies than is the non-generative representation.

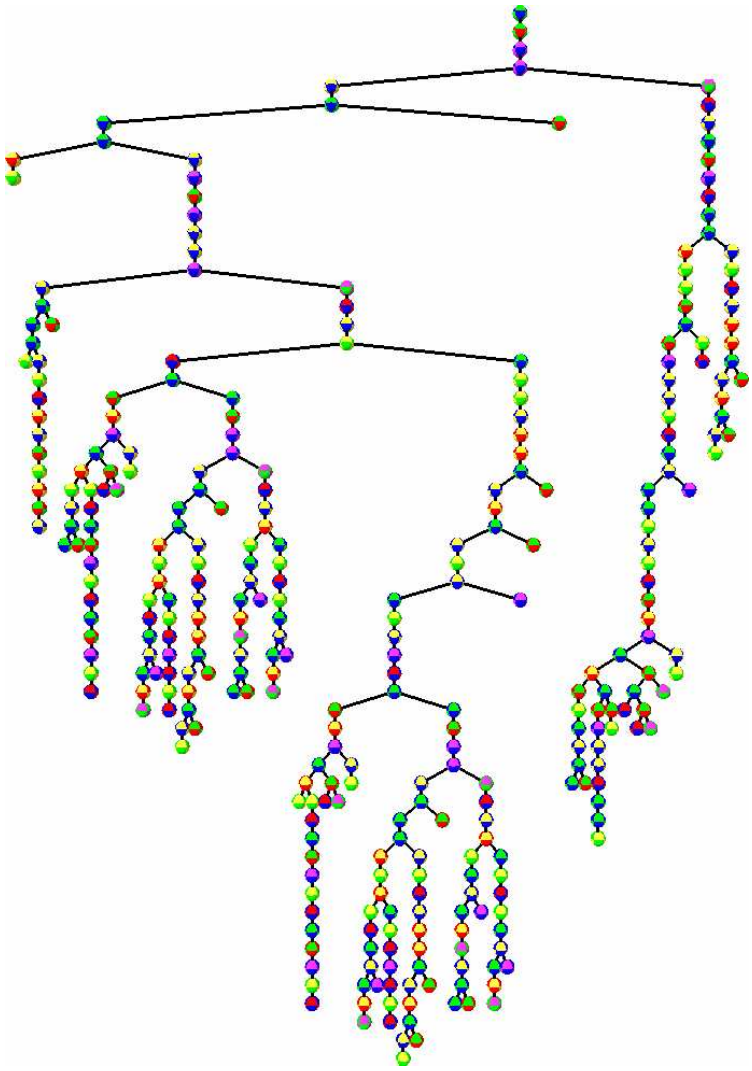
7.2 Future Work

The work of this thesis has opened the door to several directions for future work:

- The resulting assembly procedures produced with *GENRE* are linear, but the principles of this design representation apply to different structures. An obvious extension



(a)



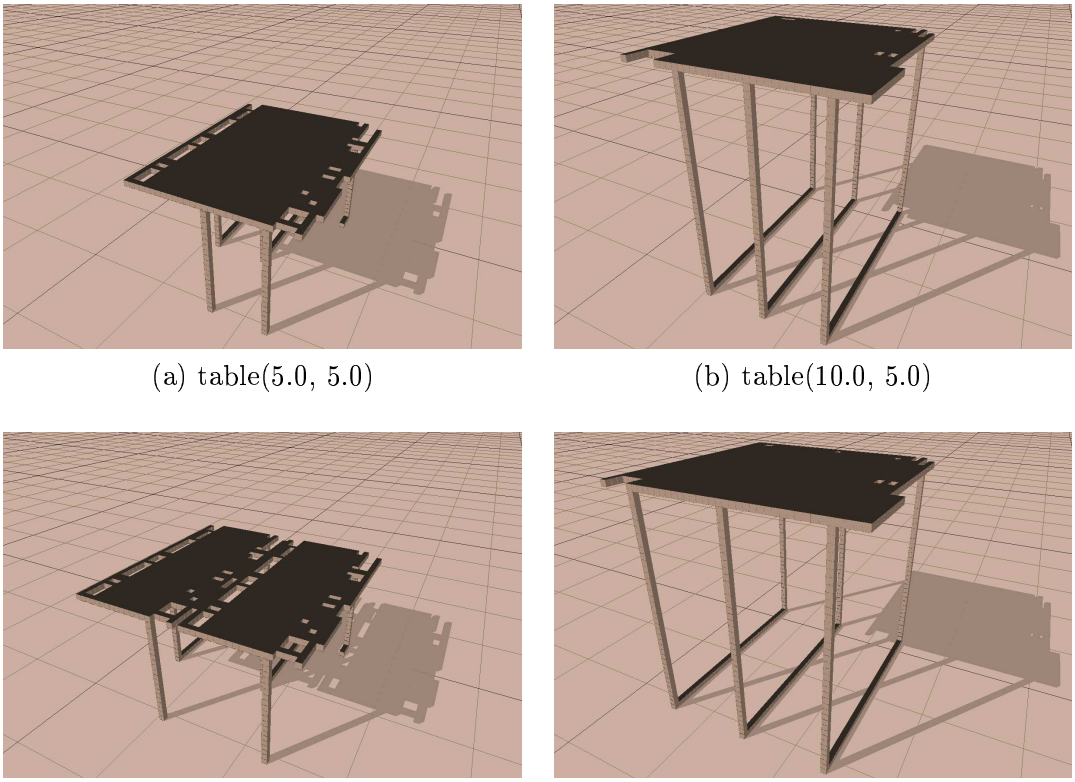
(b)

Figure 7.1: A graphical rendition of: (a), a tree-structured generative representation; and (b), the tree-structured assembly procedure it produces.

of this is to go from strings to trees, as is done with cellular encoding and genetic programming, such as in figure 7.1. This would allow evolution of genetic programming trees for comparison with work in this field as well as better enable the construction of graphs.

- Of the four different properties that design representations were identified as having, the two representations compared had either all four of these properties or none of them. Further work could investigate the usefulness of other representations with different combinations of these properties. For example, one use of parameters and conditionals is parametric design, in which an individual represents not just one instance of a design but an entire class of designs over some parameter space. Initial work towards evolving representations for an entire class of objects was demonstrated with Gruau's cellular encoding in which manually changing a single value in the evolved encoding of a network controls the size of the resulting network for solving different sizes of n-parity problem [51]. By utilizing the starting parameters to the initial production call, the generative representation described here can describe a parametric class of objects; the example of section 4.1 shows two instances of designs for a parametric class of tree-like objects in figure 4.2.

  Figures 7.2 and 7.3 are examples of parameter-controlled designs produced with *GENRE*. The four images in figure 7.2 are four tables produced by the same evolved table encoding. The parameters for height and surface/volume were scaled so that there was a non-direct relationship between the input parameters and the desired property value. Input values for height were 5.0 and 10.0, for which the desired height values were 20 and 40. Input values for surface/volume were also 5.0 and 10.0, for which the desired surface areas were 400 and 1600 and desired volumes were 8000 and 64000. The three networks shown in figure 7.3 are generated with inputs 3.0, 5.0 and 7.0 to the same individual and correctly solve the 3/5/7-parity problem of section 5.2.1. An analysis of this individual (see appendix B) finds that the generative representation used some



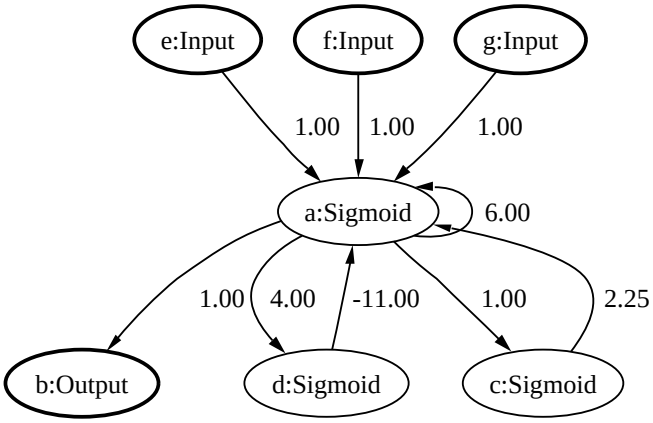
(a) table(5.0, 5.0)

(b) table(10.0, 5.0)

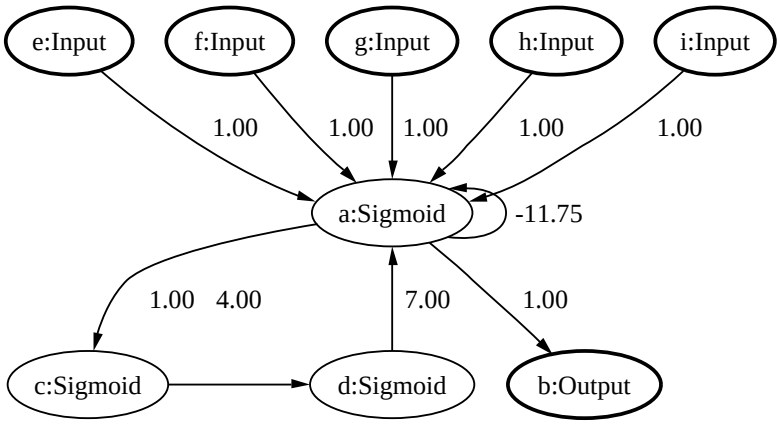
(c) table(5.0, 10.0)

(d) table(10.0, 10.0)

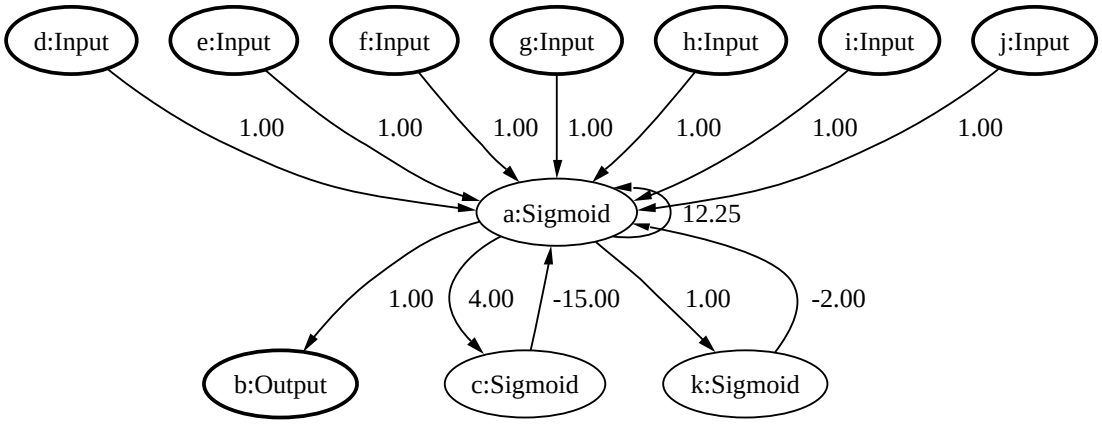
Figure 7.2: Four different tables constructed using different parameter values with the same design encoding.



(a) 3-parity



(b) 5-parity



(c) 7-parity

Figure 7.3: Networks constructed to solve 3, 5, 7-parity from the same evolved network encoding.

sections of the genotype for constructing parts of all three networks and used conditionals so that other sections of the genotype were used for constructing only one of the networks.

- The generative representation described in this thesis uses a procedural approach of explicitly constructing designs through an assembly procedure. An alternative to using explicit commands for constructing a design is to use a representation in which the encoded design rules interact to construct a design implicitly, such as with cellular automata and and artificial chemistries [31]. A procedural approach has two strengths which are not readily apparent with non-procedural representations, both of which are examples of modularity, a principle of good software design [92].

  First, an explicit, procedural representation allows for a modularization of construction rules for different parts of the artifact into different parts of the representation. This allows for the hierarchical construction of assemblies of parts which, through encapsulation, can be transfered from one individual to another as a unit allowing the variation operator to scale. In addition, this allows for reused components to be duplicated in the genotype so that they can be adapted into another task. Both of these operations would be difficult with an implicit representation because of the indirect association between a given element of the genotype and a part in the artifact and the high interaction between rules in creating a design.

  A second advantage with a procedural approach is the decomposition between the encoding and the construction of a design results in a generic design system. Different classes of design can be evolved by changing the set of construction commands and the design builder [110]. For example, in this thesis the same evolutionary algorithm and generative representation were used for all four design classes. Changing a non-procedural representation to work on the different design classes of chapter 4 would have required a redesign of the representation and variation operators because of the high integration between representation and design.

- Finally, even though evolutionary algorithms were used as the search algorithm for optimizing artifact designs, the arguments for the advantages of generative representations did not depend on the specific search algorithm used. Another direction for future work would be to investigate the advantages of generative representations over non-generative representations with other search algorithms, such as hill-climbing and simulated annealing.

The next step in automated design is in producing design representations that can hierarchically create and reuse assemblies of parts in ever more powerful ways. 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 systems.

Appendix A

An Evolved Table

The following L-system is the generative representation for the table in figure A.1, and also shown in figure 6.1.a:

P0 (n1>10.0) :- P11(n0/4.0,2.0) down(1.0) {P17(3.0,n1/2.0) P12(n1+n0,n0+n1) P3(1.0,n1-n0) }(4.0)

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

(n1>0.0) :- [P16(2.0,n0+2.0)]

P2 (n0>5.0) :- P7(n1/2.0,n0+2.0) back(1.0)

(n0>5.0) :- P7(n1/2.0,n0+2.0) back(1.0)

(n0>0.0) :- [left(4.0) P12(3.0,4.0)]

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

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

(n0>0.0) :- down(1.0) {clockwise(n1) forward(3.0) }(5.0)

P6 (n1>1.0) :- [back(5.0) left(1.0) back(5.0) down(1.0) up(1.0) back(5.0) back(5.0) back(5.0)]

(n0>1.0) :- [back(5.0) up(1.0) back(n0) left(1.0) back(n0) down(5.0) up(1.0) back(5.0) left(n1) back(5.0) down(1.0)]

(n1>0.0) :- [back(5.0) left(1.0) back(5.0) down(1.0) up(1.0) back(5.0) back(5.0) back(5.0)]

P7 (n0>-1.0) :- [clockwise(1.0) clockwise(1.0) left(1.0) clockwise(1.0) clockwise(5.0) right(1.0) clockwise(1.0)] down(1.0)
(n1>1.0) :- [clockwise(1.0) clockwise(1.0) left(1.0) clockwise(1.0) clockwise(5.0) clockwise(1.0) clockwise(1.0)] down(1.0)
(n0>0.0) :- [clockwise(1.0) left(1.0)] right(2.0) up(2.0) P18(n1-5.0,n1+4.0)

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

P9 (n1>3.0) :- P7(3.0-3.0,n0+n1) clockwise(1.0) P8(n1-n0,n1+1.0)
(n1>2.0) :- P7(3.0-3.0,n0+n1) clockwise(1.0) P8(n1-n0,n1+1.0)
(n1>0.0) :- forward(1.0) P16(n1-1.0,n0-n1)

P11 (n1>4.0) :- [P19(4.0,5.0)]
(n1>-10.0) :- right(1.0)
(n1>0.0) :-

P12 (n1>4.0) :- back(1.0) up(1.0)
(n1>4.0) :- back(1.0) up(1.0)
(n1>0.0) :- counter-clockwise(4.0)

P14 (n1>3.0) :- P11(n1,n0/n1)
(n0>10.0) :- P2(n1/3.0,n1+n0) P9(n1,n0/n1)
(n1>0.0) :- P9(n1,n0/n1)

P16 (n1>22.0) :-
(n1>5.0) :-
(n1>0.0) :- [P19(n0/2.0,n1-n0) back(1.0) forward(3.0)] clockwise(2.0) P3(5.0,n1-5.0)

P17 (n1>3.0) :- up(n1) back(2.0) back(4.0) back(n0) back(3.0) back(3.0) back(5.0)
(n1>3.0) :- up(n1) back(4.0) back(2.0) back(2.0) back(2.0) back(3.0) back(5.0)
(n1>0.0) :- up(2.0) back(2.0)

P18 (n1>3.0) :- back(n1) P14(n1-3.0,3.0) left(1.0) left(1.0) right(1.0) right(1.0)

(n1>3.0) :- back(n1) P14(n1-2.0,3.0) left(1.0) [counter-clockwise(1.0)] right(1.0)

(n0>0.0) :- [P13(5.0,n1/4.0) right(1.0)] right(1.0)

This L-system is started with P0(4, 10) and run for 17 iterations. The images in figure A.1 show the intermediate stages in the development of a table design. The generative representation for this table (see appendix A) starts with the command $P0(4.0, 10.0)$ and goes through 17 iterations of parallel replacement. The first iteration produces the string, *P11(1,2) down(1) {P17(3,5) P18(14,14) P3(1,6) }(4)*, which uses block replication to encode the table's four legs. Within this block, the productions $P17$, $P18$ and $P3$ are called once, and this is the only time they are called. The structure of the base is encoded in productions $P17$ and $P18$. Reducing the number of voxels created from its *back*() command, in these productions reduces the width and depth of the table. $P3$ calls $P16$, but all of $P16$'s conditions fail, and this sequence of productions produces no build commands. From $P18$ there are calls to $P14$ then $P9$ – $P14$ also calls $P2$ which then calls $P12$, but none of these productions produce bricks. $P9$ changes the direction of the turtle to build the table legs with the help of $P7$, and then the table legs and surface are encoded in productions $P6$ and $P8$, which construct the legs and surface through repeated calls to each other. In both $P6$ and $P8$ the parameter values are used to select which production body to use. The height of the legs is encoded in the first successor, for which the condition succeeds both when $P8$ is initially called and also the first time it calls itself. In later calls to $P8$, the first condition fails and the second condition succeeds resulting in the first call to $P6$, which begins the sequence of commands for constructing the table's surface. Later evolution changed the production rules $P6$ and $P8$ to,

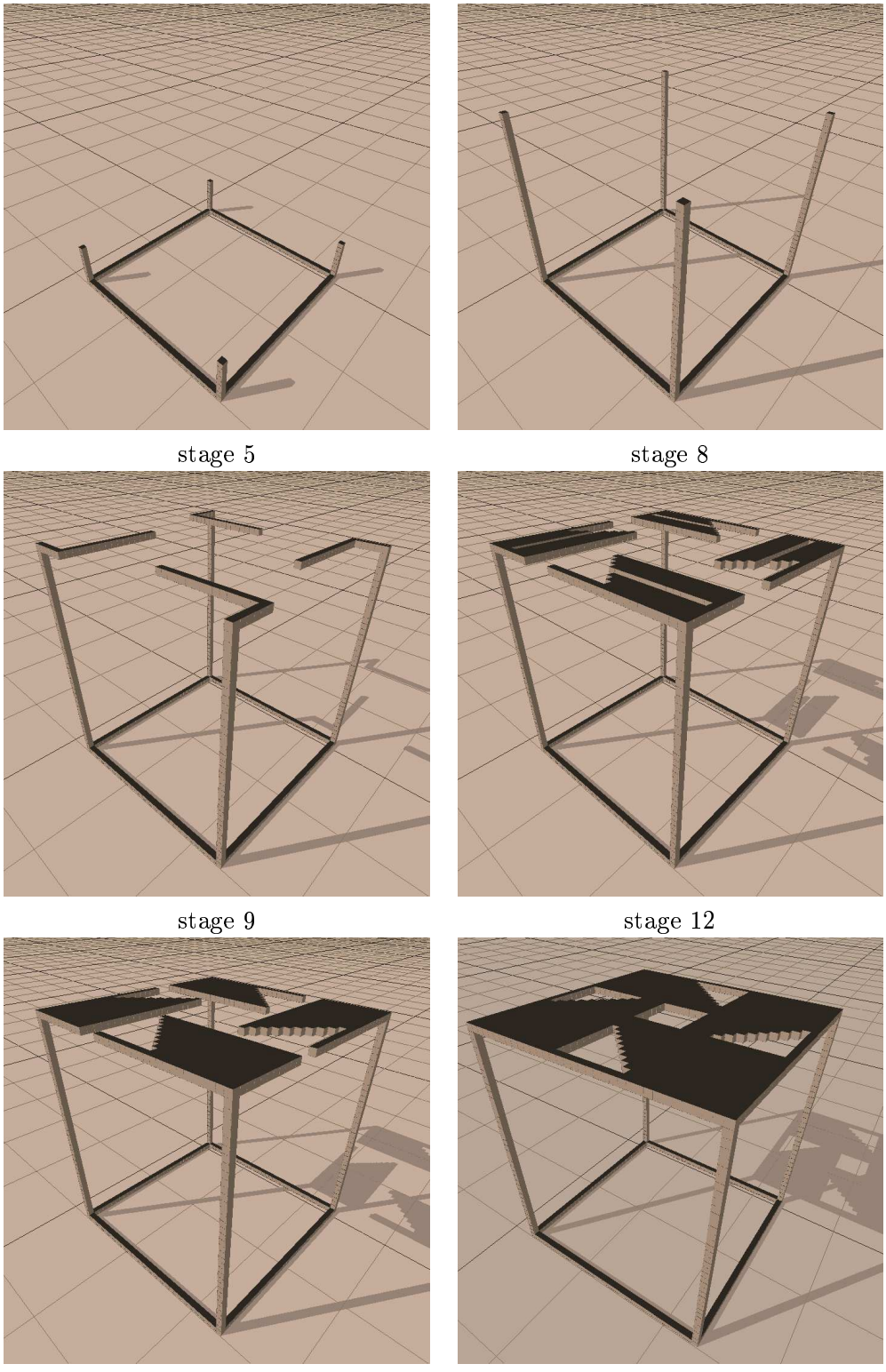


Figure A.1: Stages in the development of a table.

P6 (n1 > 1.0) → [back(5.0) up(5.0) back(n0) left(1.0) back(5.0) back(5.0) back(5.0) back(4.0)]

(n0 > 2.0) → [back(5.0) up(5.0) back(n0) back(5.0) left(1.0) down(5.0) up(5.0) back(5.0) back(5.0) back(5.0) back(4.0)]

(n1 > 0.0) → [back(5.0) up(5.0) left(1.0) back(n0) back(5.0) down(5.0) up(1.0) back(5.0) back(5.0) back(5.0) back(4.0)]

P8 (n0 > 0.0) → P8(n0/5.0,n1+1.0) [back(4.0) back(4.0) P8(n1-2.0,n0-5.0)]

(n1 > -2.0) → [P8(n0/4.0,n1+1.0) back(5.0) back(4.0) P8(2.0-5.0,3.0-5.0) back(4.0) back(5.0) P6(n1-n0,n0+n1)]

(n0 > -1.0) → counter-clockwise(1.0) down(3.0) down(n0)

with the resulting table shown in figure 5.7.a. Thus through the process of evolution the generative representation has evolved an encoding which has a modularization of the table into the number of legs, height and surface.

This generative representation, when started with P0(4, 10) and run for 17 iterations, produces the following assembly procedure:

right(1) down(1) up(5) back(2) back(4) back(3) back(3) back(3) back(5) back(14) [left(4) counter-clockwise(4)] [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1) clockwise(1) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(5) back(4) back(4) back(4)]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(13.7) left(1) back(13.7) down(5) up(1) back(5) left(-0.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(12.7) left(1) back(12.7) down(5) up(1) back(5) left(-1.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(11.7) left(1) back(11.7) down(5) up(1) back(5) left(-2.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(10.7) left(1) back(10.7) down(5) up(1) back(5) left(-3.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(9.7) left(1) back(9.7) down(5) up(1)

back(5) left(-4.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(4.4) left(1) back(4.4) down(5) up(1) back(5) left(0.9) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(8.7) left(1) back(8.7) down(5) up(1) back(5) left(-5.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(3.4) left(1) back(3.4) down(5) up(1) back(5) left(-0.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(7.7) left(1) back(7.7) down(5) up(1) back(5) left(-6.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(2.4) left(1) back(2.4) down(5) up(1) back(5) left(-1.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(6.6) left(1) back(6.6) down(5) up(1) back(5) left(-7.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.1) left(1) back(1.1) down(5) up(1) back(5) left(0.2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.4) left(1) back(1.4) down(5) up(1) back(5) left(-2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(5.5) left(1) back(5.5) down(5) up(1) back(5) left(-8.2) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] left(1) left(1) right(1) right(1) up(5) back(2) back(4) back(3) back(3) back(3) back(5) back(14) [left(4) counter-clockwise(4)] [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1) clockwise(1) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(5) back(4) back(4) back(4)]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(13.7) left(1) back(13.7) down(5) up(1) back(5) left(-0.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(12.7) left(1) back(12.7) down(5) up(1) back(5) left(-1.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(11.7) left(1) back(11.7) down(5) up(1) back(5) left(-2.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(10.7)

left(1) back(10.7) down(5) up(1) back(5) left(-3.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(9.7) left(1) back(9.7) down(5) up(1) back(5) left(-4.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(4.4) left(1) back(4.4) down(5) up(1) back(5) left(0.9) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(8.7) left(1) back(8.7) down(5) up(1) back(5) left(-5.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(3.4) left(1) back(3.4) down(5) up(1) back(5) left(-0.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(7.7) left(1) back(7.7) down(5) up(1) back(5) left(-6.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(2.4) left(1) back(2.4) down(5) up(1) back(5) left(-1.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(6.6) left(1) back(6.6) down(5) up(1) back(5) left(-7.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.1) left(1) back(1.1) down(5) up(1) back(5) left(0.2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.4) left(1) back(1.4) down(5) up(1) back(5) left(-2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(5.5) left(1) back(5.5) down(5) up(1) back(5) left(-8.2) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] left(1) left(1) right(1) right(1) up(5) back(2) back(4) back(3) back(3) back(3) back(5) back(14) [left(4) counter-clockwise(4)] [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1) clockwise(1) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [back(5) back(4) back(4) back(4)]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(13.7) left(1) back(13.7) down(5) up(1) back(5) left(-0.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(12.7) left(1) back(12.7) down(5) up(1) back(5) left(-1.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)

] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(11.7) left(1) back(11.7) down(5) up(1) back(5) left(-2.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(10.7) left(1) back(10.7) down(5) up(1) back(5) left(-3.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(9.7) left(1) back(9.7) down(5) up(1) back(5) left(-4.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(4.4) left(1) back(4.4) down(5) up(1) back(5) left(0.9) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(8.7) left(1) back(8.7) down(5) up(1) back(5) left(-5.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(3.4) left(1) back(3.4) down(5) up(1) back(5) left(-0.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(7.7) left(1) back(7.7) down(5) up(1) back(5) left(-6.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(2.4) left(1) back(2.4) down(5) up(1) back(5) left(-1.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(6.6) left(1) back(6.6) down(5) up(1) back(5) left(-7.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.1) left(1) back(1.1) down(5) up(1) back(5) left(0.2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.4) left(1) back(1.4) down(5) up(1) back(5) left(-2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(5.5) left(1) back(5.5) down(5) up(1) back(5) left(-8.2) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] left(1) left(1) right(1) right(1) up(5) back(2) back(4) back(3) back(3) back(3) back(5) back(14) [left(4) counter-clockwise(4)] [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1) clockwise(1) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(5) back(4) back(4) back(4)]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5)

up(1) back(13.7) left(1) back(13.7) down(5) up(1) back(5) left(-0.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(12.7) left(1) back(12.7) down(5) up(1) back(5) left(-1.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(11.7) left(1) back(11.7) down(5) up(1) back(5) left(-2.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(10.7) left(1) back(10.7) down(5) up(1) back(5) left(-3.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(9.7) left(1) back(9.7) down(5) up(1) back(5) left(-4.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(4.4) left(1) back(4.4) down(5) up(1) back(5) left(0.9) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(8.7) left(1) back(8.7) down(5) up(1) back(5) left(-5.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(3.4) left(1) back(3.4) down(5) up(1) back(5) left(-0.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(7.7) left(1) back(7.7) down(5) up(1) back(5) left(-6.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(2.4) left(1) back(2.4) down(5) up(1) back(5) left(-1.1) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(6.6) left(1) back(6.6) down(5) up(1) back(5) left(-7.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[[[[[[[[[[[[[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) [back(4) back(4)] [back(4) back(4)] back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.1) left(1) back(1.1) down(5) up(1) back(5) left(0.2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(1.4) left(1) back(1.4) down(5) up(1) back(5) left(-2) back(5) down(1)]] back(5) back(4) back(4) back(4) [back(5) up(1) back(5.5) left(1) back(5.5) down(5) up(1) back(5) left(-8.2) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4)

back(4)] [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] [back(4) back(4)]] left(1) left(1) right(1) right(1)

Appendix B

A Neural-Network for 3/5/7-Parity

This appendix contains the genotype for the parametric parity-network constructor referred to in section 5.2. The command set used in this encoding is listed in table 4.2.

P0 (n0>7.0) → output(1.0) parent(1.0) P11(n0+n1,n0+n1) parent(1.0) P10(n0,n0/n1) split(1.0) decrease-weight(1.0) reverse(n1) merge(1.0)

(n0>6.0) → output(1.0) parent(1.0) P11(n0+n1,n0+n1) parent(1.0) P10(n0,n0/n1) split(1.0) loop(1.0) reverse(n1) merge(1.0) merge(n1)

(n1>0.0) → output(1.0) parent(1.0) P11(n0+n1,n0+n1) parent(1.0) split(1.0) P10(n0,n0/n1) reverse(n1) merge(1.0)

P1 (n0>6.0) → {increase-weight(1.0) input(1.0) P3(n1-1.0,n1/4.0) parent(1.0) parent(1.0) }(n0) loop(1.0) [decrease-weight(3.0)]

(n0>5.0) → {increase-weight(1.0) input(1.0) P3(n1-1.0,n1/4.0) parent(1.0) loop(1.0) }(n0)

(n0>2.0) → {increase-weight(1.0) input(1.0) P3(n1-1.0,n1/4.0) parent(1.0) parent(1.0) }(n0) loop(1.0) [decrease-weight(3.0)]

P3 (n1>4294967294.0) → {parent(4.0) duplicate(3.0) }(n0) merge(1.0)

(n1>4294967293.0) → {decrease-weight(1.0) parent(4.0) duplicate(3.0) }(n0) merge(1.0)

(n1>0.0) → {parent(4.0) duplicate(3.0) parent(4.0) duplicate(3.0) duplicate(3.0) }(n0) merge(1.0)

P4 (n0>3.0) → next(1.0)

(n0>-1.0) → next(1.0)

(n1>2.0) → parent(1.0) parent(1.0)

P9 (n0>12.0) → parent(1.0) duplicate(1.0)

(n1>10.0) → parent(1.0) duplicate(1.0)

(n1>0.0) → parent(1.0) duplicate(1.0)

P10 (n1>0.0) → [increase-weight(1.0) parent(4.0) next(1.0) decrease-weight(2.0) parent(4.0) next(1.0) split(4.0)] P1(n0-0.0,n1/1.0) reverse(1.0)

(n0>3.0) → parent(1.0) [decrease-weight(2.0) parent(4.0) next(1.0) split(4.0)] {increase-weight(1.0) P3(n1-1.0,n1-4.0) input(1.0) parent(1.0) loop(1.0) }(n0) reverse(1.0)

(n0>0.0) → split(1.0)

P11 (n0>12.0) → [parent(1.0) P4(n0-n1,5.0) reverse(1.0) duplicate(1.0)] decrease-weight(n1) parent(3.0) parent(1.0) loop(1.0) reverse(4.0) duplicate(2.0)

(n1>10.0) → [P9(n0-0.0,n1-0.0)] decrease-weight(1.0) parent(3.0) parent(1.0) loop(1.0) reverse(4.0) merge(2.0) merge(1.0) decrease-weight(n0) duplicate(2.0)

(n1>0.0) → [P4(n0-n1,5.0) parent(1.0) reverse(1.0) duplicate(1.0)] decrease-weight(n1) parent(1.0) loop(1.0) reverse(4.0) duplicate(2.0)

This L-system is started with either P0(3, 6), P0(5, 6) or (P0(7, 6) and it produces a network with 3, 5 or 7 inputs and 1 output that correctly solves the 3, 5 or 7 parity problem. The final assembly procedure generated with P0(3.0,6.0) is:

add-output(1.0) parent(1.0) [next(1.0) parent(1.0) reverse(1.0) duplicate(1.0)] decrease-weight(9.0) parent(1.0) loop(1.0) reverse(4.0) duplicate(2.0) parent(1.0) split(1.0) [increase-weight(1.0) parent(4.0) next(1.0) decrease-weight(2.0) parent(4.0) next(1.0) split(4.0)] increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) loop(1.0) [decrease-weight(3.0)] reverse(1.0) reverse(6.0) merge(1.0)

The final assembly procedure generated with P0(5.0,6.0) is:

add-output(1.0) parent(1.0) [parent(1.0) duplicate(1.0)] decrease-weight(1.0) parent(3.0) parent(1.0) loop(1.0) reverse(4.0) merge(2.0) merge(1.0) decrease-weight(11.0) duplicate(2.0) parent(1.0) split(1.0) [increase-weight(1.0) parent(4.0) next(1.0) decrease-weight(2.0) parent(4.0) next(1.0) split(4.0)] increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) loop(1.0) [decrease-weight(3.0)] reverse(1.0) reverse(6.0) merge(1.0)

The final assembly procedure generated with P0(7.0,6.0) is:

add-output(1.0) parent(1.0) [parent(1.0) next(1.0) reverse(1.0) duplicate(1.0)] decrease-weight(13.0) parent(3.0) parent(1.0) loop(1.0) reverse(4.0) duplicate(2.0) parent(1.0) [increase-weight(1.0) parent(4.0) next(1.0) decrease-weight(2.0) parent(4.0) next(1.0) split(4.0)] increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) increase-weight(1.0) add-input(1.0) merge(1.0) parent(1.0) parent(1.0) loop(1.0) [decrease-weight(3.0)] reverse(1.0) split(1.0) loop(1.0) reverse(6.0) merge(1.0) merge(6.0)

The networks constructed with these assembly procedures are shown in figure 7.3.

Appendix C

Sorting Programs

This appendix describes work using *GENRE* to evolve computer programs. In these experiments, only the generative reprentation is used because the field of genetic programming has already shown that representations with reuse achieve higher fitness in fewer generations then non-generative representations [82].

C.1 Sorting Program Construction Language

Table C.1: List of commands for sorting programs.

Command	Description
[ ]	If FLAG is set, execute instructions enclosed within.
clear	Clears FLAG.
compare(n)	Compares data elements in memory locations n and $n+1$ and sets FLAG if $n < n+1$.
swap(n)	Swaps data elements in memory locations n and $n+1$.

Sorting programs are different from the other classes of problems in that the assembly procedure is the program. Instead of construction commands, the non-production rules in the generative representation are commands for a kind of assembly language for a virtual computer that consists of a FLAG register and a random access memory. These commands are listed in table C.1. The brackets, '[' and ']', are used as a conditional on the FLAG

register. If the FLAG register is set then the commands inside the brackets are executed, otherwise these commands are skipped. *Clear* unsets the command register. *Compare(n)* compares the values in memory locations n and $n+1$. If the value in memory location n is less than the value in memory location $n+1$ this command sets the FLAG register. The only command that affects data is *swap(n)*, which exchanges the values in memory locations n and $n+1$. So that compare() and swap() do not access memory elements outside the range used, their input parameter is modulus the size of the memory array.

C.2 Sorting Program Results

This problem domain consists of finding a linear sequence of computer instructions that correctly sort all 128 binary inputs of length seven. The instruction set is described in section C.1 and is listed in table C.1. Individuals were evaluated by running with all 128 inputs and fitness was the sum of scores for each input vector.

C.2.1 Fitness for Sorting Programs

The graph in figure C.1 plots the fitness of the best individual in the population, averaged over 25 trials, with the generative representations.

C.2.2 Reuse and Evolvability for the Sorting Programs

The graph in figure C.2.a plots the average length of the genotype for the generative representations as well as plotting the average length of the assembly procedure produced by the generative representation. From this graph it can be seen that with the generative representation, the number of data elements in the encoding of a sorting program are used approximately twenty times, on average, in creating a sorting program.

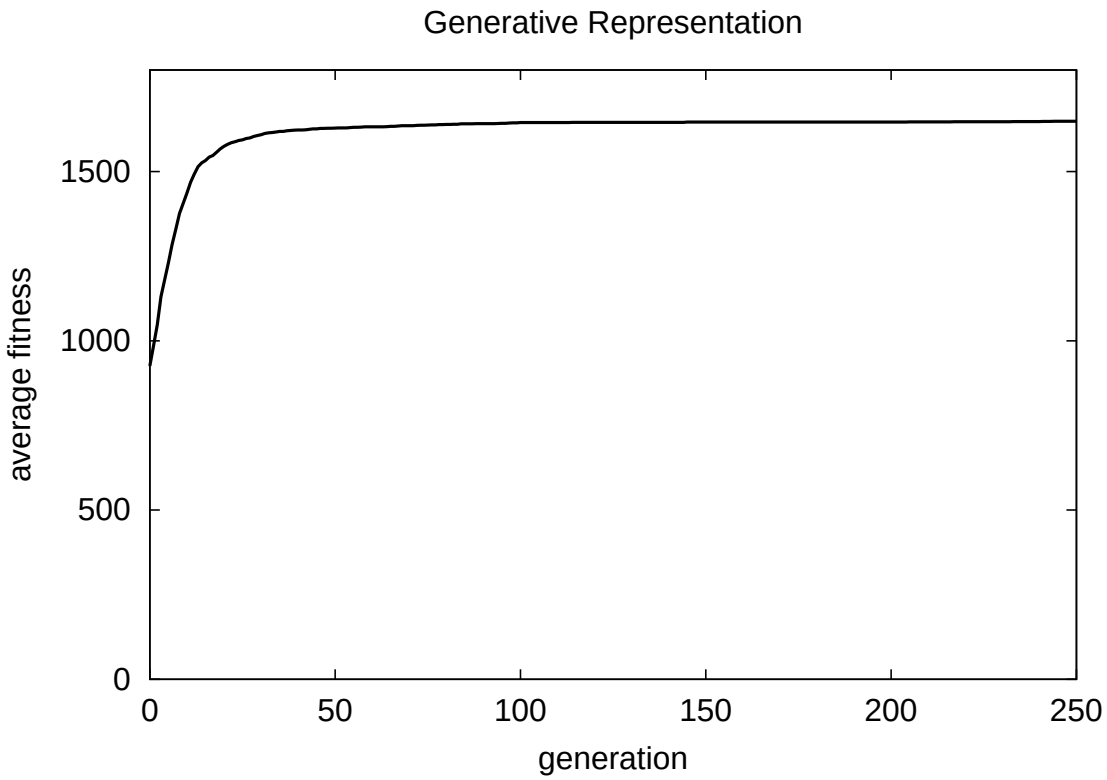


Figure C.1: Performance for the evolution of sorting programs using the generative representation.

Table C.2: Summary of results for sorting programs.

Summary of results	Generative
Average final best fitness	1649
Percentage of runs which find solution.	64%
Average generation solution was found (of runs that found a solution).	82

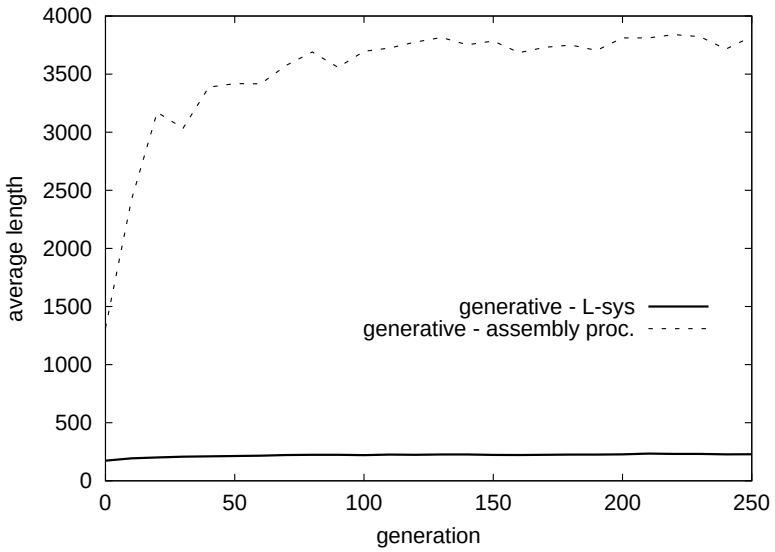


Figure C.2: Graph of length of the generative representation genotype and the assembly procedure it produced.

C.3 Summary of Results for the Sorting Programs

The experiments for evolving sorting programs are summarized in table C.2. One shortcoming with the existing evolutionary design system is it is limited to evolving linear assembly procedures. Extending this to tree structured genotypes would allow for future work to compare the results of this system with other work in genetic programming.

Appendix D

Two-dimensional Robots

The first version of the system for evolving genobots used a two-dimensional world. Actuated joints cycle through 60° with variable frequency and relative phase offset. Static joints accommodate rods at fixed 60° angles. No comparisons between direct and generative representations where performed with it, as the three-dimensional environment was implemented shortly thereafter.

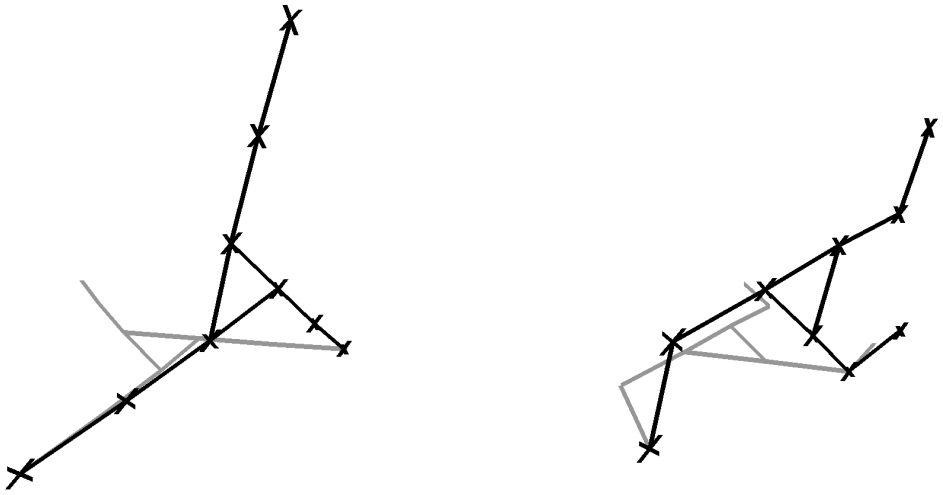
D.1 Two-dimensional Robot Construction Language

Command	Description
[ ]	Push/pop state to stack.
forward	Add bar.
back	Move backwards to previous rod.
joint(n)	Forward, end with an actuated joint which moves at speed n.
clockwise(n)	Rotate heading clockwise $n \times 60°$.
counter-clockwise(n)	Rotate heading counterclockwise $n \times 60°$.
increase-offset(n)	Increase phase offset by $n \times 25\%$.
decrease-offset(n)	Decrease phase offset by $n \times 25\%$.

Table D.1: Design language for oscillator-controlled, two-dimensional robots.

The assembly procedure consists of a sequence of build commands that give instructions to a LOGO-style turtle that is used to construct a robot from rods. Commands are listed

in table D.1. [and] push and pop the current state - consisting of the current bar, the orientation and joint oscillation offset - to and from a stack. Forward moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar. Back goes backwards up the parent of the current bar. Clockwise and counter-clockwise rotate the orientation in steps of 60°. The joint command operates the same as forward except that if a new bar is created, it ends with an actuated joint which oscillates over a range of 60°. The parameter to this command specifies the speed at which the joint oscillates, using integer values from 1 to 5, and the relative offset of the oscillation cycle is taken from the turtle's state. Increase-offset and decrease-offset change the offset value in the turtle's state by ±25% of a total cycle.



(a) (b)

Figure D.1: A sample oscillator-controlled, two-dimensional genobot.

An example genobot constructed by these commands is shown in figure D.1. It is created from the string,

joint(1) [joint(1) forward] clockwise(2) joint(1) [joint(1) forward] clockwise(2)

joint(1) [joint(1) forward] clockwise(2)

X's are used to show the location of actuated joints. The left image shows the robot with all actuated joints in their starting orientation and the image on the right shows the same

robot with all actuated joints at the other extreme of their actuation cycle. In this example all actuated joints are moving in phase.

D.2 Generative Representation Example for Oscillator Controlled Genobots

The following is a design encoding using the generative representation and the command set listed in table D.1. It consists of two productions with each production containing two condition-successor pairs:

$$\begin{aligned} P0(n): \quad & n > 2 \rightarrow \{\ P0(n-1)\ \}(n) \\ & n > 0 \rightarrow joint(1)\ P1(n \times 2)\ clockwise(2) \end{aligned}$$

$$\begin{aligned} P1(n): \quad & n > 2 \rightarrow [\ P1(n/4)\ ] \\ & n > 0 \rightarrow joint(1)\ forward \end{aligned}$$

If the design encoding is started with P0(3), the resulting sequence of strings is produced:

1. *P0(3)*
2. *{P0(2) }(3)*
3. *{ joint(1) P1(4) clockwise(2) }(3)*
4. *{ joint(1) [P1(1)] clockwise(2) }(3)*
5. *{ joint(1) [joint(1) forward] clockwise(2) }(3)*
6. *joint(1) [joint(1) forward] clockwise(2) joint(1) [joint(1) forward] clockwise(2) joint(1) [joint(1) forward] clockwise(2)*

This final string produces the genobot in figure D.1.

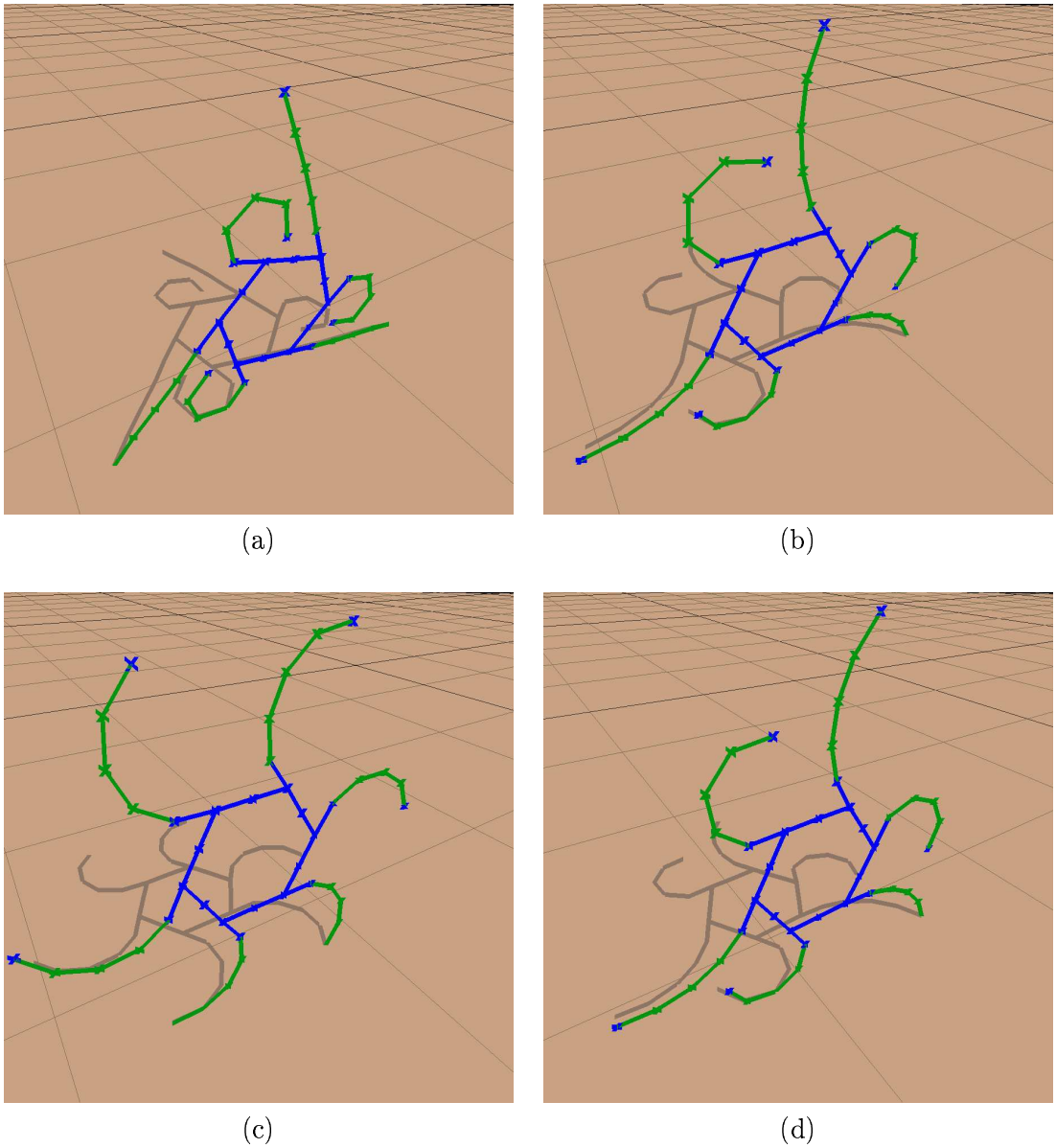


Figure D.2: The locomotion cycle of a walking star creature, built from 43 bars and 24 actuated joints.

D.3 Evolved Two-dimensional Genobots

To create designs we run our evolutionary algorithm with 100 individuals for a maximum of 500 generations. The P0L-systems used have fifteen productions with each production having two parameters and three sets of condition-successor pairs. Fitness is a function of the distance moved by the creature's center of mass after 10 simulated seconds.

Evolutionary runs were similar in many ways. The first individuals would mainly move their center of mass while remain in a fixed location. Typically individuals with a repeating locomotion cycle would appear by generation 30 and these would compete to be the dominant design in the population. Once converged to one design the EA would try many variations of it making slow and steady improvements. Periodically a new version would have a significantly greater fitness and then the population would converge to this variant.

Each evolutionary run was different, resulting in a variety of different creatures, but all creatures could be classed into one of the following families: crawlers who would use one appendage to drag the rest of the body forward; walkers who used legs to move with little dragging; inch-worms that inched along; and rollers that rotated their whole body to move, see figures D.2 and D.3.

An example of an actual robot constructed from an evolved design is the walking M in figure 5.27.a. This robot moves by bringint its two outer arms, which are 25% out of phase, together to lift its middle arm and then to shift its center of mass to the right. One modification to the constructed robot is the addition of sandpaper on the feet of the two outer arms to compensate for the friction modeled by our simulator and that of the actual surface used.

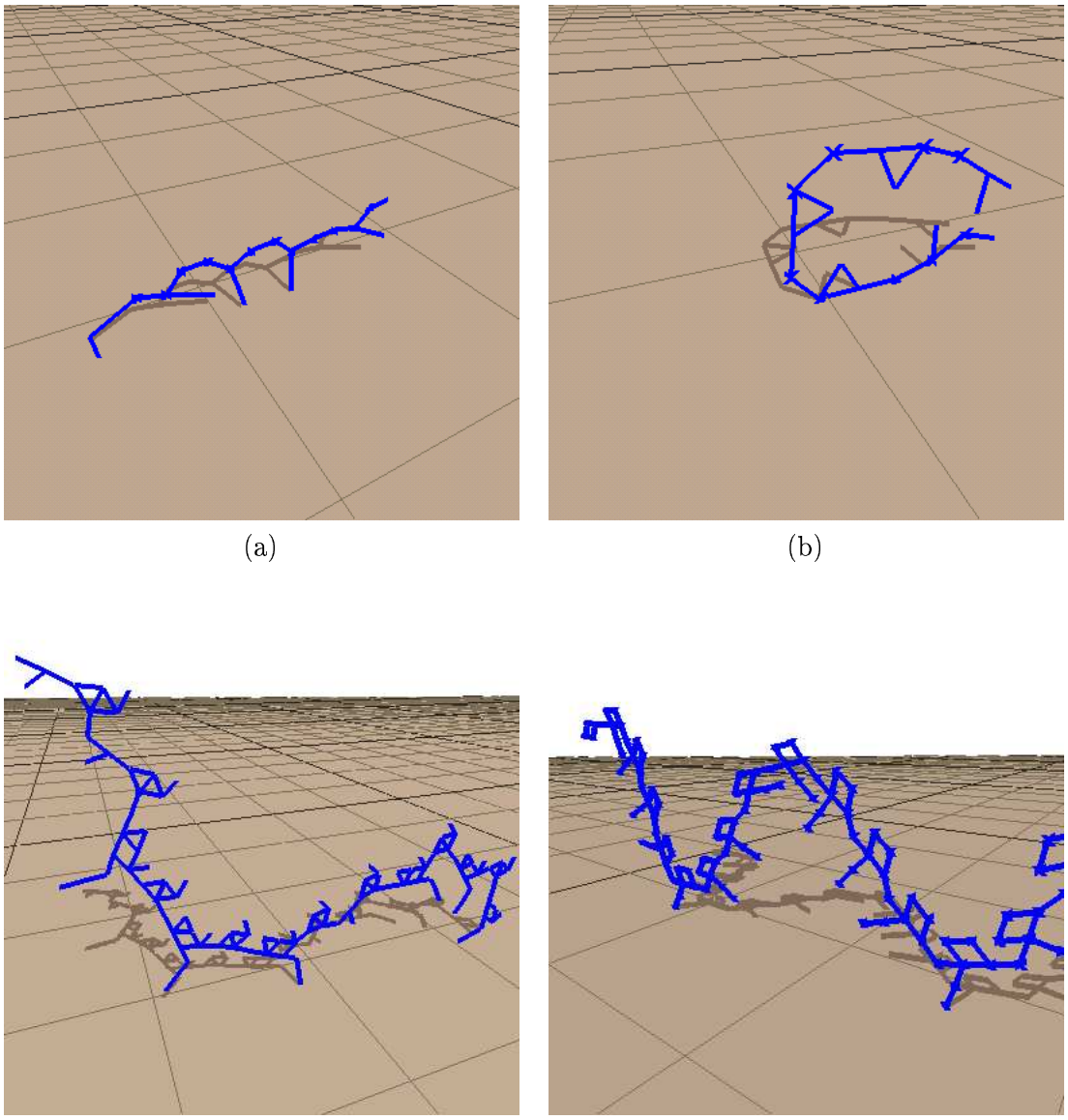


Figure D.3: Evolved creatures: *a*, a multi-legged walker with 24 bars and 12 actuated joints; *b*, a rolling circle with 23 bars and 6 actuated joints; *c*, an inch-worm built from 143 bars and 16 actuated joints; and *d*, an undullating serpent with 164 bars and 61 actuated joints; Notice the re-use of components.

Appendix E

An Oscillator-Controlled Genobot

The genotype for the genobot in figure 5.27.b (called the Kayak) is:

P0 (n1>6.0) → increase-offset(n1) P3(5.0,n1-5.0) [down(1.0) P7(n0,n0-1.0)]
(n0>6.0) → increase-offset(3.0) P3(5.0,n1-5.0) [counter-clockwise(n0) P12(n0,n0-1.0) counter-clockwise(n0) P12(n0,n0-1.0)]
(n1>0.0) → increase-offset(3.0) P3(5.0,n1-5.0) [down(1.0) counter-clockwise(n0) down(1.0) counter-clockwise(n0) P12(n0,n0-1.0)]

P1 (n1>5.0) → [back] P1(n1/4.0,3.0) clockwise(1.0)
(n0>4.0) → {revolute2(n0) }(2.0) right(1.0) up(1.0) [up(1.0) down(5.0) decrease-offset(1.0)]
(n1>1.0) → {revolute2(n0) }(2.0) right(1.0) up(1.0) [up(1.0) down(5.0) down(1.0)] [up(1.0) down(4.0) decrease-offset(1.0)]

P2 (n1>0.0) → up(1.0) P12(n1/5.0,n1/n0) revolute(1.0)
(n1>1.0) → up(1.0) P14(n1/5.0,n1/n0) revolute(1.0)

P3 (n1>-3.0) → left(4.0) P10(n0-3.0,n0-1.0) P2(n0/5.0,n0-5.0) P1(n1-1.0,1.0) left(4.0)
(n0>3.0) → left(4.0) P10(n0-3.0,n0-1.0) P2(n0/5.0,n0-5.0) left(4.0)
(n1>4.0) → decrease-offset(1.0) P2(n0/5.0,n0-5.0)

P4 (n1>4.0) → P6(n0+n1,5.0) counter-clockwise(n0) up(1.0) P13(n1,n1/5.0) P1(4.0,5.0) right(1.0) up(n1)

(n0>4.0) → counter-clockwise(n0) P6(n0+n1,5.0) up(1.0) P13(n1,n1/5.0) P1(4.0,5.0) right(1.0) up(n1) up(n1)

(n0>0.0) → counter-clockwise(n0) P6(n0+n1,5.0) up(1.0) P13(n1,n1/5.0) P1(4.0,5.0) right(1.0) up(n0) up(n1)

P5 (n0>-1.0) → right(3.0) right(3.0) P4(n0+4.0,4.0)

(n0>0.0) → increase-offset(3.0) right(3.0) P4(n0+4.0,4.0)

(n1>0.0) → right(3.0) right(3.0) P4(n0+4.0,4.0)

P6 (n0>8.0) → [P9(2.0-3.0,n0-1.0) P10(n0+n1,n1-2.0)]

(n0>-3.0) → [P9(2.0-3.0,n0-1.0) P10(n0+n1,n1-2.0)]

(n1>0.0) → clockwise(n0) counter-clockwise(n1)

P7 (n1>0.0) → left(1.0)

(n0>0.0) → {{down(1.0) }(n0) }(4.0)

P8 (n0>5.0) → counter-clockwise(1.0) counter-clockwise(3.0) up(1.0)

(n0>1.0) → clockwise(2.0) P14(2.0,n0+1.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)

(n1>0.0) → [clockwise(2.0) P14(2.0,n0+1.0)]

P9 (n1>5.0) → up(5.0)

(n1>1.0) → P5(n0-1.0,n0/5.0) P7(n1/n0,5.0/4.0) P7(n1/n0,n1/1.0)

(n0>0.0) → P5(n0-1.0,n0/5.0)

P10 (n1>4.0) → [decrease-offset(3.0) forward] [clockwise(1.0)]

(n1>3.0) → right(1.0) {decrease-offset(5.0) P11(3.0,n1+n0) P6(n0-5.0,n0-4.0) }(2.0)

(n1>0.0) → right(1.0) {decrease-offset(5.0) P6(n0-5.0,n0-4.0) P11(4.0,n1+n0) }(2.0) decrease-offset(5.0)

P11 (n0>1.0) → counter-clockwise(1.0) P11(n0/4.0,n0+n1)

(n0>0.0) → counter-clockwise(1.0) P9(n1/4.0,n0-1.0) P12(n1/5.0,5.0)

(n0>0.0) → counter-clockwise(1.0) P12(n1/5.0,5.0)

P12 (n1>1.0) → left(1.0) [back right(1.0)] P9(n1-4.0,3.0) [decrease-offset(1.0) P8(n1,n1+n0)] left(n0)

(n1>-1.0) → left(1.0) [back down(n0)] P9(n1-4.0,3.0) left(n1)

(n1>0.0) → [back down(n0)] [P8(n0,n0-n1)]

P13 (n0>4.0) → left(1.0) P12(4.0,1.0-2.0)

(n0>3.0) → [[decrease-offset(1.0)] clockwise(4.0) decrease-offset(1.0) left(1.0)] left(1.0) P12(4.0,1.0-2.0)

(n1>0.0) → [] left(1.0) clockwise(4.0)

P14 (n0>5.0) → clockwise(1.0) P5(n1+n0,n1/5.0)

(n0>0.0) → [clockwise(1.0)] clockwise(1.0)

Started with P0(2, 6) and run for 19 iterations this genotype produces the following assembly procedure:

increase-offset(3.0) left(4.0) right(1.0) decrease-offset(5.0) counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) counter-clockwise(5.2) [up(5.0) right(1.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) left(1.0) [back right(1.0)] [decrease-offset(1.0) clockwise(2.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(4.2) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) left(1.0) [back right(1.0)] [decrease-offset(1.0) clockwise(2.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(4.2) decrease-offset(5.0)] up(1.0) [[decrease-offset(1.0)] clockwise(4.0) decrease-offset(1.0) left(1.0)] left(1.0) revolute2(4.0) revolute2(4.0) right(1.0) up(1.0) [up(1.0) down(5.0) down(1.0)] [up(1.0) down(4.0) decrease-offset(1.0)] right(1.0) up(4.0) up(4.0) left(1.0) [back right(1.0)] right(3.0) right(3.0) counter-clockwise(4.0) [up(5.0) right(1.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) left(1.0) [back right(1.0)] [decrease-offset(1.0)] left(4.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) left(1.0) [back right(1.0)] [decrease-offset(1.0)] left(4.0) decrease-offset(5.0)] up(1.0) [[decrease-offset(1.0)] clockwise(4.0) decrease-offset(1.0) left(1.0)] left(1.0) revolute2(4.0) revolute2(4.0) right(1.0) up(1.0) [up(1.0) down(5.0) down(1.0)] [up(1.0) down(4.0) decrease-offset(1.0)] right(1.0) up(4.0) up(4.0) left(1.0) left(1.0) [decrease-offset(1.0) clockwise(2.0) [clockwise(1.0)] clockwise(1.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(1.8) decrease-offset(5.0) counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) counter-clockwise(5.2) [up(5.0) right(1.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) left(1.0) [back right(1.0)] [decrease-offset(1.0) clockwise(2.0)

counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(4.2) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) right(3.0) right(3.0) left(1.0) [back right(1.0)] [decrease-offset(1.0) clockwise(2.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(4.2) decrease-offset(5.0)] up(1.0) [[decrease-offset(1.0)] clockwise(4.0) decrease-offset(1.0) left(1.0)] left(1.0) revolute2(4.0) revolute2(4.0) right(1.0) up(1.0) [up(1.0) down(5.0) down(1.0)] [up(1.0) down(4.0) decrease-offset(1.0)] right(1.0) up(4.0) up(4.0) left(1.0) [back right(1.0)] right(3.0) right(3.0) counter-clockwise(4.0) [up(5.0) right(1.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) left(1.0) [back right(1.0)] [decrease-offset(1.0)] left(4.0) decrease-offset(5.0) [up(5.0) [decrease-offset(3.0) forward] [clockwise(1.0)]] counter-clockwise(1.0) counter-clockwise(1.0) left(1.0) [back right(1.0)] [decrease-offset(1.0)] left(4.0) decrease-offset(5.0)] up(1.0) [[decrease-offset(1.0)] clockwise(4.0) decrease-offset(1.0) left(1.0)] left(1.0) revolute2(4.0) revolute2(4.0) right(1.0) up(1.0) [up(1.0) down(5.0) down(1.0)] [up(1.0) down(4.0) decrease-offset(1.0)] right(1.0) up(4.0) up(4.0) left(1.0) left(1.0) [decrease-offset(1.0) clockwise(2.0) [clockwise(1.0)] clockwise(1.0) counter-clockwise(1.0) counter-clockwise(1.0) up(1.0)] left(1.8) left(4.0) [down(1.0) counter-clockwise(2.0) down(1.0) counter-clockwise(2.0) left(1.0) [back down(2.0)] left(1.0) left(1.0) left(1.0)]

Appendix F

A Neural-Network Controlled Genobot

This appendix contains the genotype for the neural-network controlled genobot shown in figure 6.4.b. The command set used by this genobot is listed in table 4.6.

P0 (n0>5.0) → reverse(1.0) left(5.0) merge(1.0) increase-weight(1.0) back(1.0) back(1.0) duplicate(1.0) up(1.0) decrease-weight(4.0) [loop(4.0) P19(1.0,n1+4.0) duplicate(1.0)] counter-clockwise(1.0) forward(5.0) reverse(1.0)

(n1>3.0) → [counter-clockwise(1.0) left(3.0) P10(n1,1.0)] [clockwise(1.0) left(3.0) split(1.0) split(1.0)] P19(n1-4.0,n0/2.0) forward(1.0) counter-clockwise(1.0)

(n0>4.0) → [counter-clockwise(1.0) left(3.0) P10(n1,1.0)] [P6(n0-0.0,n1-0.0)] up(1.0) P19(n1-4.0,n0/2.0) forward(1.0) counter-clockwise(1.0)

P1 (n0>5.0) → [parent(1.0) loop(1.0) forward(1.0) merge(1.0)] [forward(3.0) increase-weight(1.0) increase-weight(1.0) clockwise(n0) left(1.0)]

(n0>5.0) → [parent(1.0) loop(1.0) forward(1.0) merge(1.0)] [forward(3.0) increase-weight(1.0) increase-weight(1.0) clockwise(n0) left(1.0)]

(n1>0.0) → [split(1.0) clockwise(1.0) parent(1.0)] forward(1.0) split(1.0) counter-clockwise(1.0) right(5.0) forward(1.0)

P6 (n0>5.0) → clockwise(1.0) left(3.0) split(1.0) split(1.0)

(n1>3.0) → clockwise(1.0) left(3.0) split(1.0) split(1.0)

(n0>0.0) → clockwise(1.0) left(3.0) split(1.0) split(1.0)

P8 (n0>5.0) → [merge(1.0) back(4.0)]

(n0>-1.0) → increase-weight(3.0) P1(n0-4.0,n0+n1) split(1.0) parent(1.0) merge(1.0)

(n1>0.0) → increase-weight(3.0) P1(n0-4.0,n0+n1) split(1.0) parent(1.0) merge(1.0) increase-weight(3.0) P1(n0-4.0,n0+n1) split(1.0) parent(1.0) merge(1.0)

P10 (n1>5.0) → (right(n1) set-function(n1) decrease-weight(1.0) duplicate(1.0) down(1.0)) revolute2(1.0)

(n0>2.0) → {forward(3.0) increase-weight(1.0) down(1.0) reverse(1.0) }(4.0) down(1.0) decrease-weight(1.0) P14(n1-n0,2.0) back(4.0) split(1.0) P18(n0-2.0,n0/2.0) clockwise(1.0) counter-clockwise(1.0) P10(1.0,5.0)

(n0>0.0) → {forward(1.0) down(1.0) reverse(1.0) increase-weight(1.0) }(4.0) down(1.0) decrease-weight(1.0) P14(n1-n0,2.0) back(4.0) split(1.0) P18(n0-2.0,n0/2.0) clockwise(1.0) counter-clockwise(1.0) P10(1.0,5.0)

P14 (n1>5.0) → loop(1.0) reverse(4.0) down(1.0) loop(1.0) revolute2(3.0) {forward(1.0) P16(n1/n0,n0-4.0) parent(1.0) }(n1) [parent(5.0) down(1.0) right(1.0) next(1.0) parent(4.0)]

(n0>2.0) → back(1.0) loop(1.0) loop(n0) increase-weight(1.0) clockwise(1.0)

(n0>-2.0) → back(1.0) loop(1.0) loop(4.0) increase-weight(1.0) clockwise(1.0)

P15 (n1>1.0) → counter-clockwise(1.0) counter-clockwise(1.0) counter-clockwise(1.0) decrease-weight(1.0)

(n1>1.0) → counter-clockwise(1.0) counter-clockwise(1.0) counter-clockwise(1.0) counter-clockwise(1.0)

(n1>0.0) → counter-clockwise(1.0) parent(4.0) forward(1.0) [forward(3.0) forward(1.0) increase-weight(1.0) right(1.0)] set-function(1.0) back(1.0) next(4.0) back(1.0)

P18	(n1>1.0) →	next(3.0) revolute2(1.0) forward(1.0) reverse(1.0) parent(1.0) revolute2(n0)
	(n1>0.0) →	{P8(n0-0.0,n1-0.0) next(3.0) forward(n0) next(n1) revolute2(1.0) next(3.0) revolute2(1.0) }(n1) forward(1.0) reverse(1.0) parent(1.0) revolute2(n0)
	(n0>0.0) →	{P8(n0-0.0,n1-0.0) next(3.0) revolute2(1.0) }(n1) forward(1.0) reverse(1.0) parent(1.0) revolute2(n0)
P19	(n1>1.0) →	down(1.0) merge(1.0) merge(1.0) down(1.0) decrease-weight(1.0) {merge(3.0) right(n0) right(n0) }(2.0) {right(n0) right(n0) }(2.0) P15(n0-0.0,5.0-0.0) (n0) left(1.0)
	(n1>1.0) →	down(1.0) merge(1.0) down(1.0) decrease-weight(1.0) parent(1.0) merge(3.0) left(1.0)
	(n1>0.0) →	down(1.0) merge(1.0) down(1.0) decrease-weight(1.0) parent(1.0) merge(3.0) left(5.0)

This L-system is started with the command, P0(5, 3) and after 11 iterations of rewrites it produces the following assembly procedure,

[counter-clockwise(1) left(3) forward increase-weight(1) down(1) reverse(1) forward increase-weight(1) down(1) reverse(1) forward increase-weight(1) down(1) reverse(1) forward increase-weight(1) down(1) reverse(1) down(1) decrease-weight(1) backward split(1) next(3) revolute2(1) forward reverse(1) parent(1) revolute2(1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1)

loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward loop(1) loop(4) increase-weight(1) clockwise(1) backward split(1) forward reverse(1) parent(1) revolute2(-1) clockwise(1) counter-clockwise(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) forward down(1) reverse(1) increase-weight(1) down(1) decrease-weight(1) backward split(1) clockwise(1) counter-clockwise(1)] [clockwise(1) left(3) split(1) split(1)] up(1) down(1) merge(1) merge(1) down(1) decrease-weight(1) left(1) forward counter-clockwise(1)

Appendix G

Source Code

GENRE consists of over 50,000 lines of C++ code spread across more than 50 files. These files describe the evolutionary algorithm, L-system representation (with compiler and variation operators), two-dimensional and three-dimensional physics simulators, statistics module, linear algebra and quaternion modules, neural network module, graphics module, random number generator, and other components of the evolutionary design system. This appendix contains some of the source code for the implementation of the generative-representation compiler described in chapter 3.

Each individual in *GENRE* is an L-system and is defined by the C++ class `ind_lsys`, which uses the class `prod_body` for each production body. The following sections are the header files and parts of the source code for compiling an L-system to its assembly procedure and also for storing the execution history of this compilation process. Not included is the source code for creating random L-systems or for mutating/recombining L-systems.

An individual's L-system is compiled into an assembly procedure in `ind_lsys`'s procedure `evaluate_helper()`. In this procedure, `start_tape()` is called to start the compilation process, and then each iteration of rewriting is performed by calling the procedure `process_tape()`. The final string is converted to an assembly procedure by calling the procedure `tape_to_string()`.

G.1 ind_lsys.hh

```
#define CREATE_INIT -1
#define CREATE_MUTATE 0
#define CREATE_RECOMBINE 1
#define TAPE_END '.'

#define NUM_PRODUCTIONS 15
#define MAX_PROD_BODIES 2
#define MAX_PROD_VARS 3
#define MAX_CONSTANTS 10
#define MAX_FUNCTIONS 0

#define OP_RECOMB 1
#define OP_MUTATE 2
#define OP_REC_RULE 0
#define OP_REC_PROD 1
#define NUM_RECOMB_OPS 2
#define OP_MUT_CONST 2
#define OP_MUT_CHAR 3
#define OP_MUT_ADD 4
#define OP_MUT_DEL 5
#define OP_MUT_REORDER 6
#define OP_MUT_COND 7
#define OP_MUT_REPLICATE 8
#define NUM_MUTATE_OPS 7

#define OP_REC_CONST 9
#define OP_UNKNOWN 9

extern Real Rec_delta;
extern int Num_arguments;
extern int Num_constants;
extern int Num_functions;
extern int Num_productions;
extern int Max_bodies;
extern int Num_operations;
extern Real Prob_mult_seq;
extern Real Prob_branch_edge;

extern int Max_prod_length;
extern int Prod_length;
extern int Max_string_length;

class Prod_Body;
```

```
typedef struct History
{
  int index;
  int how_created;
  int operation;
  int num_mutate;
  int num_recombine;
  int op_count[15];
  Real delta;
  Real val1;

  Real fitness[10];
  Real fitness_prev[10];
} history;

typedef struct tConstant
{
  Real value;
  int used;
} Constant;

typedef struct tTape
{
  char type;
  char symbol;
  Real *arg;
} Tape;

typedef struct Production_Head
{
  int num_bodies;
  int used;
  Real *max;
  Real *min;
  Prod_Body **body;
} *Prod_Head;

class Individual
{
public:
  int created;
  int evaluated;
  int version;
  int iterations;
```

```
int length;
Constant constant[MAX_CONSTANTS];
Prod_Head *rules;
LString *lstring;
ModelStruct model;

history hist_self;
history hist_parent1;
history hist_parent2;

Individual(void); // perhaps input data struct
Individual(Individual *sample);
void clear(void);
int randomize_rule(int r);
void make_random(void);
void init(void);
void init(int init_h);
void init_pheno(int init_h); // For making single production rule & body.
int repair_rule(Prod_Head ph);
int repair_rule(int i);
void repair_init(void);

int create_mutate(void);
int better(void);
int better(Individual *other);
int replace(Individual *other);
int fitness_fail(void);
Real get_fitness(void);
void get_fitness(Real *fitness_vector);
void set_fitness(Real val);
Real get_fitness(int index);
void set_fitness(Real *val);
void set_fitness(int index, Real val);
void add_fitness(Real val);
void div_fitness(Real val);
void set_creation(int type);
int get_creation(void);
void init_hist(history *hist);
void init_hist(void);
void duplicate_hist(history *hist_from, history *hist_to);
void duplicate(Individual *ind);
int duplicate_pheno(Individual *ind);
void op_used(int op_num);
int system_length(void);
```

```
int duplicate_used_bodies(int r);
int mutate_rule_duplicate(int r);
int mutate_rule_new(int r);
int mutate_rule(int r);
void clear_rule(int r);
int random_used_body_p2(int r);
int random_used_body(int r);
int random_unused_body(int r);
int is_rule_used(int r);
int random_used_rule(void);
int random_unused_rule(void);
int random_used_constant(void);
int mutate_constant(void);
int encapsulate(int r);
int mutate(Real alpha);
int mutate(void);
int mutate_pheno(void);
int mutate_init(void);
int recombine_constants(Individual *parent2);
int recombine_productions(Individual *parent2);
int recombine_rule(Individual *parent2);
int join(Individual *parent2);
int recombine(Individual *parent2);
int recombine_pheno(Individual *parent2);

int process_tape(Tape *tape_src, Tape *tape_dest);
void reset_used(void);
int start_tape(Real *var, Tape *tape);
int tape_to_string(Tape *tape, LString *string);
int edit_distance(Individual *ind2);
int phenotype_distance(Individual *ind2);
int phenotype_distance2(Individual *ind2);
int phenotype_dist(Individual *ind2);
void display(void);
Real evaluate_helper(int init, int iterations, Real *argument);
Real evaluate(int init);
Real evaluate(void);

void print_fitness(void);
void print_history(history *hist);
void print_history2(history *hist);
void fprint_history2(history *hist, FILE *log);
void print_history_self(void);
void print_history_self2(void);
void fprint_history_self2(FILE *log);
```

```
  void print_history(void);
  void fprint_history(FILE *log);
  void print_history_full(void);
  void print_results(void);
  void info(void);

  int read_history(history *hist, FILE *file);
  int read(FILE *file);
  void write_history(history *hist, FILE *file);
  void write(FILE *file);
  void write_pheno(FILE *file);
};

void print_lchar(int i, LString *lstring);
```

G.2 ind_lsys.cc

```
extern Real evaluate_ind(int init, int size, Individual *ind);

int Num_arguments = MAX_PROD_VARS;
int Num_constants = MAX_CONSTANTS;
int Num_functions = MAX_FUNCTIONS;
int Num_productions = NUM_PRODUCTIONS;
int Max_bodies = MAX_PROD_BODIES;
int Num_operations = 10;

int Num_processed;

LString Lstring[MAX_STRING_LENGTH + BONUS_LENGTH];
Tape Tape1[MAX_STRING_LENGTH + BONUS_LENGTH];
Tape Tape2[MAX_STRING_LENGTH + BONUS_LENGTH];
int Created_tape = FALSE;

int process_rule(Real *var, Tape *tape_dest, Prod_Head phead);
int tape_2_string(char end_type, char end_symbol,
                  int *t, Tape *tape, int *s, char *string);
Real table_fitness(Real p_height, Real p_surf, Individual *ind);

// Returns the number of symbols added.
int process_rule(Real *arg, Tape *tape_dest, Prod_Head phead)
{
  int i;

  // Statistics stuff.
  phead->used++;
```

```
  for (i=0; i<Num_arguments; i++) {
    if (phead->used == 1) {
      phead->min[i] = arg[i];
      phead->max[i] = arg[i];
    } else {
      if (arg[i] < phead->min[i])
        phead->min[i] = arg[i];
      else if (arg[i] > phead->max[i])
        phead->max[i] = arg[i];
    }
  }

  // Process the rule.
  for (i=0; i<phead->num_bodies; i++) {
    if (phead->body[i]->test_cond(arg)) {
      if (Print[3])
        printf("%d:", i);
      return phead->body[i]->write_body(arg, tape_dest);
    }
  }
  return 0;
}

Individual :: Individual(void)
{
  init_hist(&hist_self);
  init_hist(&hist_parent1);
  init_hist(&hist_parent2);
  created = FALSE;
  evaluated = FALSE;
  iterations = 4;
  version = IND_VERSION;
}

Individual :: Individual(Individual *sample)
{
  init_hist(&hist_self);
  init_hist(&hist_parent1);
  init_hist(&hist_parent2);
  created = FALSE;
  evaluated = FALSE;
  version = IND_VERSION;

  if (sample != 0)
```

```
    iterations = sample->iterations;
}

void Individual :: init(int init_h)
{
  int i, j;

  if (!Created_tape) {
    Created_tape = TRUE;

    int num_args = Num_arguments;
    if (num_args == 0)
      num_args = 1;
    for (i=0; i<MAX_STRING_LENGTH+BONUS_LENGTH; i++) {
      Tape1[i].arg = (Real*)malloc(num_args * sizeof(Real));
      if (Tape1[i].arg == 0) {
        printf("Ind_lsys :: init() malloc for tape 1 failed.\n");
        return;
      }

      Tape2[i].arg = (Real*)malloc(num_args * sizeof(Real));
      if (Tape2[i].arg == 0) {
        printf("Ind_lsys :: init() malloc for tape 1 failed.\n");
        return;
      }
    }
  }

  if (!created) {
    created = TRUE;
    length = -1;
    if (Option[10]) {
      lstring = (LString*)malloc(MAX_STRING_LENGTH*sizeof(LString));
      if (lstring == 0) {
        printf("Ind_lsys :: init() malloc for lstring failed.\n");
        return;
      }

    } else {
      lstring = (LString*)malloc(MAX_STRING_LENGTH*sizeof(LString));
    }
    if (Option[10])
      model.model_id = -1;
    else
      model.model_id = 0;
```

```
model.lstring = lstring;
if (Num_fit_dims == 1) {
  model.fv_length = 1;
} else if (Num_fit_dims == 4) {
  model.fv_length = 4;
} else {
  model.fv_length = Num_fit_dims;
}
model.fitness = (Real*)malloc(Num_fit_dims * sizeof(Real));
model.build_args = (Real*)malloc(MAX_CONSTANTS * sizeof(Real));
model.offset[0] = 0.0;
model.offset[1] = 0.0;
model.offset[2] = 0.0;

rules = (Prod_Head*)malloc(Num_productions * sizeof(Prod_Head));
for (i=0; i<Num_productions; i++) {
  rules[i] = (Prod_Head)malloc(sizeof(Production_Head));
  if (rules[i] == 0) {
    printf("Ind_lsys :: init() malloc for rules[%d] failed.\n", i);
    return;
  }

  rules[i]->max = (Real*)malloc(Num_arguments * sizeof(Real));
  if (rules[i]->max == 0) {
    printf("Ind_lsys :: init() malloc for rules[%d]->max failed.\n", i);
    return;
  }

  rules[i]->min = (Real*)malloc(Num_arguments * sizeof(Real));
  if (rules[i]->min == 0) {
    printf("Ind_lsys :: init() malloc for rules[%d]->min failed.\n", i);
    return;
  }

  rules[i]->body = (Prod_Body**)malloc(Max_bodies * sizeof(Prod_Body*));
  if (rules[i]->body == 0) {
    printf("Ind_lsys :: init() malloc for rules[%d]->body failed.\n",
            i);
    return;
  }

  for (j=0; j<Max_bodies; j++) {
    rules[i]->body[j] = new Prod_Body();
    rules[i]->body[j]->phead = rules[i];
    rules[i]->body[j]->constant = &(constant[0]);
```

```
        rules[i]->body[j]->parent = this;
      }
    }
  }

  if (init_h) {
    init_hist(&hist_self);
    init_hist(&hist_parent1);
    init_hist(&hist_parent2);
  }
  created = TRUE;
}

void Individual :: init(void)
{
  init(TRUE);
}

void Individual :: init_hist(history *hist)
{
  int i;

  hist->index = -1;
  hist->how_created = CREATE_INIT;
  hist->num_mutate = 0;
  hist->num_recombine = 0;
  hist->delta = 0.0;

  for (i=0; i<Num_fit_dims; i++) {
    hist->fitness[i] = 0.0;
    hist->fitness_prev[i] = 0.0;
  }

  for (i=0; i<Num_operations; i++)
    hist->op_count[i] = 0;
}

void Individual :: init_hist(void)
{
  int i;

  hist_self.index = -1;
  hist_self.how_created = CREATE_INIT;
  hist_self.num_mutate = 0;
  hist_self.num_recombine = 0;
```

```
  hist_self.delta = 0.0;
  hist_self.fitness_prev[0] = 0.0;

  for (i=0; i<Num_operations; i++)
    hist_self.op_count[i] = 0;
}

int Individual :: process_tape(Tape *tape_src, Tape *tape_dest)
{
  int src = 0, dst = 0;

  while (tape_src[src].symbol != TAPE_END) {
    if (tape_src[src].type == PBC_TYPE_PRODUCTION) {
      // Handle production
      dst += process_rule(tape_src[src].arg, &(tape_dest[dst]),
                          rules[tape_src[src].symbol]);

    } else {
      // Handle non-production.
      tape_dest[dst].type = tape_src[src].type;
      tape_dest[dst].symbol = tape_src[src].symbol;
      tape_dest[dst].arg[0] = tape_src[src].arg[0];
      dst++;
    }
    src++;

    if (dst >= MAX_STRING_LENGTH)
      break;
  }

  tape_dest[dst].type = PBC_TYPE_NONE;
  tape_dest[dst].symbol = TAPE_END;

  return dst;
}

void Individual :: reset_used(void)
{
  int i, j;

  for (i=0; i<Num_constants; i++)
    constant[i].used = 0;

  for (i=0; i<Num_productions; i++) {
    rules[i]->used = 0;
```

```
    for (j=0; j<Num_arguments; j++) {
      rules[i]->max[j] = 0.0;
      rules[i]->min[j] = 0.0;
    }

    for (j=0; j<rules[i]->num_bodies; j++) {
      rules[i]->body[j]->used = 0;
      rules[i]->body[j]->tested = 0;
    }
  }
}

int Individual :: start_tape(Real *arg, Tape *tape_dest)
{
  int t;

  t = process_rule(arg, tape_dest, rules[0]);
  tape_dest[t].type = PBC_TYPE_NONE;
  tape_dest[t].symbol = TAPE_END;

  return t;
}

int tape_2_string(char end_type, char end_symbol, int *t,
                  Tape *tape, int *s, LString *string)
{
  char symbol, type;
  int i, mult_val, s2;

  while (1) {
    if (Num_processed++ > MAX_PROCESSED) {
      return FALSE;
    }

    if (*s >= MAX_STRING_LENGTH) {
      string[0].symbol = 0;
      return FALSE;
    }

    type = tape[*t].type;
    symbol = tape[*t].symbol;

    if ((type == end_type) && (symbol == end_symbol)) {
      // End of block.
      *t += 1;
```

```
  if (type == PBC_TYPE_BLOCK) {
    if ((symbol == SYMBOL_POP) || (symbol == SYMBOL_POP_EDGE)) {
      string[*s].symbol = symbol;
      *s += 1;
    }
  }

  return TRUE;

} else if (type == PBC_TYPE_PRODUCTION) {
  // Production.
  *t += 1;
  continue;

} else if ((type == PBC_TYPE_BLOCK) && (symbol == SYMBOL_BLK_START)) {
  // This allows handling of nested blocks.
  // Must do multiple times.
  mult_val = int(tape[*t].arg[0]);

  if (mult_val > 0) {
    int t2 = *t + 1;
    for (i=0; i<mult_val; i++) {
      *t = t2;
      if (!tape_2_string(PBC_TYPE_BLOCK, SYMBOL_BLK_END,
                         t, tape, s, string))
        return FALSE;
    }

  } else {
    s2 = *s;
    *t += 1;
    if (!tape_2_string(PBC_TYPE_BLOCK, SYMBOL_BLK_END,
                       t, tape, &s2, string))
      return FALSE;
  }

} else if ((type == PBC_TYPE_BLOCK) && (symbol == SYMBOL_PUSH)) {
  // This allows handling of nested blocks.
  string[*s].symbol = symbol;
  *s += 1;
  *t += 1;
  if (!tape_2_string(PBC_TYPE_BLOCK, SYMBOL_POP,
                     t, tape, s, string))
    return FALSE;
```

```
    } else if ((type == PBC_TYPE_BLOCK) && (symbol == SYMBOL_PUSH_EDGE)) {
      // This allows handling of nested blocks.
      string[*s].symbol = symbol;
      *s += 1;
      *t += 1;
      if (!tape_2_string(PBC_TYPE_BLOCK, SYMBOL_POP_EDGE,
                         t, tape, s, string))
        return FALSE;

    } else if (type == PBC_TYPE_TERMINAL) {
      // Build terminal.
      string[*s].symbol = symbol;
      string[*s].arg = tape[*t].arg[0];
      *s += 1;
      *t += 1;

    } else {
      printf("\nInd_lsys::tape_2_string - error processing tape: ");
      printf("->(%d:%d) end=(%d:%d) @(%d:%d)\n",
             type, symbol, end_type, end_symbol, *t, *s);
      return FALSE;
    }
  }
}

int Individual :: tape_to_string(Tape *tape, LString *string)
{
  int t=0, s=0;
  Num_processed = 0;

  if (!tape_2_string(PBC_TYPE_NONE, TAPE_END, &t, tape, &s, string)) {
    // error
    string[0].symbol = TAPE_END;
  }

  string[s].symbol = 0;
  if (string[0].symbol == TAPE_END) {
    s = 0;
  }

  model.lstring_length = s;
  model.new_string = TRUE;

  return s;
}
```

```
Real Individual :: evaluate_helper(int init, int iterations, Real *argument)
{
  int i, ok = TRUE;
  Real val;

  for (i=0; i<Num_fit_dims; i++)
    model.fitness[i] = 0.0;

  length = start_tape(argument, Tape1);
  if (length >= MAX_STRING_LENGTH) {
    ok = FALSE;
  }

  if (Option[9] && ok) {
    length = tape_to_string(Tape1, lstring);
    if (build_structure(&model)) {
      evaluate_structure(&model);
    } else {
      ;
    }
  }

  i = 1;
  while(ok) {
    if (i >= iterations)
      break;

    i++;
    length = process_tape(Tape1, Tape2);
    if (length >= MAX_STRING_LENGTH) {
      ok = FALSE;
      break;
    }
    if (Option[9]) {
      length = tape_to_string(Tape2, lstring);
      if (build_structure(&model)) {
        evaluate_structure(&model);
      } else {
        //        ok = FALSE;
        //        break;
      }
    }

    if (i >= iterations)
```

```
    break;

  i++;
  length = process_tape(Tape2, Tape1);
  if (length >= MAX_STRING_LENGTH) {
    ok = FALSE;
    break;
  }
  if (Option[9]) {
    length = tape_to_string(Tape1, lstring);
    if (build_structure(&model)) {
      evaluate_structure(&model);
    } else {
      //        ok = FALSE;
      //        break;
    }
  }
}

if (ok) {
  if (i%2)
    length = tape_to_string(Tape1, lstring);
  else
    length = tape_to_string(Tape2, lstring);

  if (length >= MAX_STRING_LENGTH) {
    ok = FALSE;
  } else {
    if (build_structure(&model)) {
      evaluate_structure(&model);
    } else {
      ok = FALSE;
    }
    if (model.error) {
      printf("Error evaluating individual.  Saving: 'last.ind'.\n");
      FILE *file1 = fopen("last.ind", "w");
      write(file1);
      fclose(file1);
      exit(-1);
    }
  }
}

if (!ok) {
  val = 0.0;
```

```
  } else if (length >= MAX_STRING_LENGTH) {
    val = 0.0;
  } else
    val = model.fitness[0];

  if (Num_fit_dims > 7) {
    set_fitness(1, model.fitness[1]);
    set_fitness(2, model.fitness[2]);
    set_fitness(3, model.fitness[3]);
    set_fitness(4, model.fitness[4]);
    set_fitness(5, model.fitness[5]);
  }

  return val;
}

Real Individual :: evaluate(int init)
{
  int i;
  Real val = 0.0, val1;
  Real attr1, attr2, penalty;
  Real argument[Num_arguments];

  for (i=0; i<Num_fit_dims; i++)
    model.fitness[i] = 0.0;

  reset_used();

  iterations = int(constant[0].value);
  constant[0].used++;
  if (iterations < 1)
    iterations = 1;

  for (i=0; i<Num_arguments; i++) {
    argument[i] = constant[i+1].value;
    constant[i+1].used++;
  }

  model.build_args[0] = constant[i+1].value;
  model.build_args[1] = constant[i+2].value;
  model.build_args[2] = constant[i+3].value;
  constant[i+1].used++;
  constant[i+2].used++;
  constant[i+3].used++;
```

```
  val = evaluate_helper(init, iterations, argument);

  if (evaluated) {
    Real val_prev;

    val_prev = get_fitness();
    hist_self.fitness_prev[0] = val_prev;
    set_fitness((val + val_prev)/2.0);
    evaluated = 2;

  } else {
    evaluated = 1;
  }

  set_fitness(val);

  if (Num_fit_dims < 4)
    Num_fit_dims = 4;

  if (Num_fit_dims == 6) {
    set_fitness(1, system_length());
    set_fitness(2, length);
    set_fitness(3, model.fitness[1]);
    set_fitness(4, model.fitness[2]);
    set_fitness(5, model.fitness[3]);

  } else if (Num_fit_dims > 7) {
    set_fitness(Num_fit_dims-2, system_length());
    set_fitness(Num_fit_dims-1, length);

  } else if (Num_fit_dims > 3) {
    set_fitness(Num_fit_dims-3, system_length());
    set_fitness(Num_fit_dims-2, length);
    set_fitness(Num_fit_dims-1, model.fitness[1]);
  }

  return model.fitness[0];
}

Real Individual :: evaluate(void) {
  return evaluate(FALSE);
}
```

G.3 prod_body

G.4 prod_body.hh

```
// Symbol types.
#define PBC_TYPE_NONE 0
#define PBC_TYPE_TERMINAL 1
#define PBC_TYPE_PRODUCTION 2
#define PBC_TYPE_BLOCK 3
#define PBC_TYPE_VALUE 4
#define PBC_TYPE_CONSTANT 5
#define PBC_TYPE_VARIABLE 6
#define PBC_TYPE_FUNCTION 7

// Operators.
#define NUM_PBC_OPS 5
#define PBC_OP_NONE 0
#define PBC_OP_ADD 1
#define PBC_OP_SUB 2
#define PBC_OP_EQUAL 3
#define PBC_OP_DIV 4
#define PBC_OP_MULT 5
#define PBC_OP_POW 6

#define OP_ADD '+'
#define OP_SUB '-'
#define OP_MULT '*'
#define OP_DIV '/'
#define OP_POW '^'
#define OP_EQUAL '='

// Comparators.
#define PBC_COMP_TRUE 0
#define PBC_COMP_LESS 1

#define COMP_LESS '<'
#define COMP_GREATER '>'

// Blocks.
#define SYMBOL_PUSH '['
#define SYMBOL_POP ']'
#define SYMBOL_BLK_START '{'
#define SYMBOL_BLK_END '}'
#define SYMBOL_PUSH_EDGE '('
#define SYMBOL_POP_EDGE ')'
```

```
typedef struct Production_Value
{
  char type;
  char symbol;
  Real value;
} Prod_Value;

typedef struct Production_Condition
{
  char comparator;
  Prod_Value arg1, arg2;
} Prod_Cond;

typedef struct Production_Parameter
{
  char op;
  Prod_Value arg1, arg2;
} Prod_Param;

typedef struct Production_Character
{
  char type;
  char symbol;
  int match;

  Prod_Param *param;
} Prod_Char;

class Prod_Body
{
public:
  /*** Variables ***/
  int tested;
  int used;

  int empty;
  Prod_Cond cond;

  int length;
  Prod_Char *string;

  Real max[MAX_PROD_VARS];
  Real min[MAX_PROD_VARS];
  Prod_Head phead;
```

```
Constant *constant; // pointer to constants in parent.
Individual *parent;

/*** Methods ***/
Prod_Body(void);
void initialize_pvalue(Prod_Value *pvalue);
void initialize_param(Prod_Param *param);
void initialize_pchar(int index);
void initialize(void);

bool is_block(int i);
bool is_block_start(int i);
bool is_mult_start(int i);
bool is_mult_end(int i);
bool is_block_end(int i);
bool is_production(int i);
bool is_terminal(int i);
int num_productions(void);

bool valid_cond(void);
bool valid(void);

void copy_symbol(int s_from, int s_to);
void move_symbol(int s_from, int s_to);
int insert_space(int s_from, int len);
int symbol_test(int index);

int delete_single_symbol(int index);
int delete_block(int index, int del_inside);
int delete_symbol(int index, int del_inside);
void delete_sequence(int l1, int l2);
void delete_string(void);

int cond_repair_pvalue(char v, Prod_Value *pval);
int make_cond_smaller_old1(void);
int make_cond_smaller(void);
int make_cond_larger(void);
int repair_cond(void);
int repair(void);
void make_random_production(int i);
void random_single_param(int i, int p, int deflt_flag);
void random_full_param(int i, int p);
void make_random_terminal(int i);
void make_random_sequence(int start, int length);
void make_random_branch(int start, int length);
```

```
void make_random_branch_edge(int start, int length);
void make_random_mult(int start, int length);

void duplicate_pchar(int to_loc, int from_loc, Prod_Body *from_body);
void duplicate_string(Prod_Body *body);
void duplicate_cond(Prod_Body *body);
void duplicate(Prod_Body *body);
int duplicate(int len, LString *string);
void set_cond_true(void);
void set_cond_greater_zero(void);
int is_cond_true(void);
int test_cond(Real *arg);
int write_body(Real *arg, Tape *tape_dest);

void mutate_pvalue(int index, Prod_Value *pvalue);
void mutate_param_single(int index);
int mutate_param_full(int index);
int mutate_param(int index);
void reset_single_param(int index);
int mutate_symbol(int index);
int mutate_type(int index);
int mutate_character_orig(void);
int mutate_character(void);
int insert_prod_rule(int i);
int add_symbol(int i);
void make_random(void);
void make_random_condition(void);

int reorder(void);
int replicate(void);
int replicate_old1(void);
int repair_arg2(void);
int usable_cond(void);
int always_true(void);
int make_valid_cond_arg2(void);
int mutate_cond_old1(void);
int mutate_cond(void);
int mutate_old1(void);
int mutate(void);
int mutate_pheno(void);

int block_beginning(int i);
int block_ending(int i);
void pick_block(int *loc1, int *loc2);
void random_block(int *l1, int *l2);
```

```
  int block_length(int l1, int l2);
  int get_loc(int start, int len);
  void random_subblock(int *l1, int *l2);

  int match_parens(char end_type, char end_symbol, int i);
  int match_parens(void);

  int recombine_blocks(Prod_Body *body);
  int recombine(Prod_Body *body);

  int fprint_char(FILE *file, int i);
  int fprint(FILE *file);

  int read(FILE *file);
  int read_old(FILE *file);
};
```

G.5 prod_body.cc

```
int Max_prod_length = MAX_PROD_LENGTH;
int Prod_length = PROD_LENGTH;
int Max_string_length = MAX_STRING_LENGTH;
int Num_pbc_ops = NUM_PBC_OPS;

#define PVALUE_TYPE 1
#define PVALUE_MAX_CREATE 10
int Pvalue_type = 0;
int Pvalue_max_create = PVALUE_MAX_CREATE;

int is_term_char(char c);
int Num_args = 1;

int is_term_char(char c)
{
  int i;
  for (i=0; i<Num_terminals; i++)
    if (c == Terminal[i])
      return TRUE;
  return FALSE;
}

int is_op(char c)
{
  if (c == OP_ADD)
    return TRUE;
```

```
  else if (c == OP_SUB)
    return TRUE;
  else if (c == OP_EQUAL)
    return TRUE;
  else if (c == COMP_GREATER)
    return TRUE;
  else if (c == COMP_LESS)
    return TRUE;
  else if (c == OP_MULT)
    return TRUE;
  else if (c == OP_DIV)
    return TRUE;
  else if (c == OP_POW)
    return TRUE;
  return FALSE;
}

Prod_Body :: Prod_Body(void)
{
  length = 0;
  used = 0;
  initialize();
}

void Prod_Body :: initialize_pvalue(Prod_Value *pvalue)
{
  pvalue->type = PBC_TYPE_NONE;
  pvalue->symbol = 0;
  pvalue->value = 0.0;
}

void Prod_Body :: initialize_param(Prod_Param *param)
{
  param->op = PBC_OP_NONE;
  initialize_pvalue(&(param->arg1));
  initialize_pvalue(&(param->arg2));
}

void Prod_Body :: initialize_pchar(int index)
{
  int i;

  string[index].type = PBC_TYPE_NONE;
  string[index].symbol = 0;
  string[index].match = 0;
```

```
  string[index].param = (Prod_Param*)malloc(Num_args*sizeof(Prod_Param));
  if (string[index].param == 0) {
    printf("Prod_Body :: initialize_pchar() malloc failed.\n");
    return;
  }

  for (i=0; i<Num_args; i++)
    initialize_param(&(string[index].param[i]));
}

void Prod_Body :: initialize(void)
{
  int i;

  length = 0;
  Num_args = Num_arguments;
  if (Num_args == 0)
    Num_args = 1;

  string = (Prod_Char*)malloc((Prod_length+10) * sizeof(Prod_Char));
  for (i=0; i<Prod_length; i++)
    initialize_pchar(i);
}

bool Prod_Body :: is_block(int i)
{
  if (string[i].type == PBC_TYPE_BLOCK)
    return TRUE;
  return FALSE;
}

bool Prod_Body :: is_block_start(int i)
{
  if (string[i].type == PBC_TYPE_BLOCK) {
    if ((string[i].symbol == SYMBOL_PUSH) ||
        (string[i].symbol == SYMBOL_BLK_START) ||
        (string[i].symbol == SYMBOL_PUSH_EDGE))
      return TRUE;
  }
  return FALSE;
}

bool Prod_Body :: is_mult_start(int i)
{
```

```
  if (string[i].type == PBC_TYPE_BLOCK) {
    if (string[i].symbol == SYMBOL_BLK_START)
      return TRUE;
  }
  return FALSE;
}

bool Prod_Body :: is_mult_end(int i)
{
  if (string[i].type == PBC_TYPE_BLOCK) {
    if (string[i].symbol == SYMBOL_BLK_END)
      return TRUE;
  }
  return FALSE;
}

bool Prod_Body :: is_block_end(int i)
{
  if (string[i].type == PBC_TYPE_BLOCK) {
    if ((string[i].symbol == SYMBOL_POP) ||
        (string[i].symbol == SYMBOL_BLK_END) ||
        (string[i].symbol == SYMBOL_POP_EDGE))
      return TRUE;
  }
  return FALSE;
}

bool Prod_Body :: is_production(int i)
{
  if (string[i].type == PBC_TYPE_PRODUCTION)
    return TRUE;
  return FALSE;
}

bool Prod_Body :: is_terminal(int i)
{
  if (string[i].type == PBC_TYPE_TERMINAL)
    return TRUE;
  return FALSE;
}

int Prod_Body :: num_productions(void)
{
  int i;
  int num = 0;
```

```
  for (i=0; i<length; i++) {
    if (is_production(i))
      num++;
  }
  return num;
}

void copy_pvalue(Prod_Value *pv_to, Prod_Value *pv_from)
{
  pv_to->type = pv_from->type;
  pv_to->symbol = pv_from->symbol;
  pv_to->value = pv_from->value;
}

void copy_param(Prod_Param *p_from, Prod_Param *p_to)
{
  p_to->op = p_from->op;
  copy_pvalue(&p_to->arg1, &p_from->arg1);
  copy_pvalue(&p_to->arg2, &p_from->arg2);
}

void copy_pchar(Prod_Char *pc_from, Prod_Char *pc_to)
{
  pc_to->type = pc_from->type;
  pc_to->symbol = pc_from->symbol;
  pc_to->match = pc_from->match;

  for (int i=0; i<Num_args; i++)
    copy_param(&pc_from->param[i], &pc_to->param[i]);
}


void Prod_Body :: copy_symbol(int s_from, int s_to)
{
  copy_pchar(&string[s_from], &string[s_to]);
}


Real calc_prod_value(Real *arg, Constant *constant, Prod_Value *pval)
{
  Real value;

  if (pval->type == PBC_TYPE_NONE)
    return 0.0;
```

```
  else if (pval->type == PBC_TYPE_VALUE)
    value = pval->value;
  else if (pval->type == PBC_TYPE_VARIABLE)
    value = arg[pval->symbol];
  else if (pval->type == PBC_TYPE_CONSTANT) {
    value = constant[pval->symbol].value;
    constant[pval->symbol].used++;

  } else {
    printf("calc prod value:: error, invalid arg type: %d.\n", pval->type);
    while (1) ;
  }
  return value;
}

Real calc_prod_param(Real *arg, Constant *constant, Prod_Param *param)
{
  Real val1, val2;

  if (param->op == PBC_OP_NONE)
    return calc_prod_value(arg, constant, &(param->arg1));

  if (param->op == PBC_OP_EQUAL)
    return calc_prod_value(arg, constant, &(param->arg2));

  val1 = calc_prod_value(arg, constant, &(param->arg1));
  val2 = calc_prod_value(arg, constant, &(param->arg2));

  if (param->op == PBC_OP_ADD)
    return val1 + val2;
  else if (param->op == PBC_OP_SUB)
    return val1 - val2;
  else if (param->op == PBC_OP_MULT)
    return val1 * val2;
  else if (param->op == PBC_OP_DIV) {
    if (absv(val2) <= 1.0) {
      // Don't allow division to increase size of value.
      return val1; // treat as if val2 = 1.
    }
    return val1 / val2;
  } else if (param->op == PBC_OP_POW)
    return pow(val1, val2);

  return 0.0;
}
```

```
int Prod_Body :: is_cond_true(void)
{
  return (cond.comparator == PBC_COMP_TRUE);
}

int Prod_Body :: test_cond(Real *arg)
{
  int i;
  Real val1, val2;

  tested++;
  if (tested == 1) {
    for (i=0; i<Num_arguments; i++) {
      min[i] = arg[i];
      max[i] = arg[i];
    }
  } else {
    for (i=0; i<Num_arguments; i++) {
      if (arg[i] < min[i])
        min[i] = arg[i];
      else if (arg[i] > max[i])
        max[i] = arg[i];
    }
  }

  if (cond.comparator == PBC_COMP_TRUE)
    return TRUE;

  val1 = calc_prod_value(arg, constant, &(cond.arg1));
  val2 = calc_prod_value(arg, constant, &(cond.arg2));

  if (cond.comparator == COMP_GREATER)
    return (val1 > val2);
  else if (cond.comparator == COMP_LESS)
    return (val1 < val2);

  return FALSE;
}

int Prod_Body :: write_body(Real *arg, Tape *tape_dest)
{
  int i, j, t = 0;

  used++;
```

```
if (used == 1) {
  for (i=0; i<Num_arguments; i++) {
    min[i] = arg[i];
    max[i] = arg[i];
  }
} else {
  for (i=0; i<Num_arguments; i++) {
    if (arg[i] < min[i])
      min[i] = arg[i];
    else if (arg[i] > max[i])
      max[i] = arg[i];
  }
}

i=0;
while (i < length) {
  tape_dest[t].type = string[i].type;
  tape_dest[t].symbol = string[i].symbol;

  if (is_production(i)) {
    for (j=0; j<Num_arguments; j++)
       tape_dest[t].arg[j] = calc_prod_param(arg, constant,
                                             &(string[i].param[j]));
  } else {
    // if multiplier block start, get arg from branch end character.
    if (is_block(i) && (string[i].symbol == SYMBOL_BLK_START)) {
       tape_dest[t].arg[0] = calc_prod_param(arg, constant,
                                &(string[string[i].match].param[0]));
      if (tape_dest[t].arg[0] == 0.0) {
        // Empty multiplier.
        if (string[i].match < i) {
          printf("Prod_body ::  Error with empty multiplier ");
          printf("%d(%c) -> %d.\n",
                 i, string[i].symbol, string[i].match);
          fprint(stdout);
        }
        i = string[i].match + 1;
        continue;
      }

    } else
       tape_dest[t].arg[0] = calc_prod_param(arg, constant,
                                             &(string[i].param[0]));
  }
```

```
    t++;
    i++;
  }

  return t;
}

void Prod_Body :: reset_single_param(int index)
{
  string[index].param[0].arg1.type = PBC_TYPE_VALUE;
  string[index].param[0].arg1.value = 1.0;
}
```

Bibliography

[1] H. Abelson and A. A. diSessa. *Turtle Geometry*. M.I.T. Press, 1982.

[2] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, second edition, 1996.

[3] M. Agarwal and J. Cagan. The language of coffee makers. *Environment and Planning B: Planning and Design*, 25(2):205–226, 1998.

[4] M. Agarwal, J. Cagan, and G. Stiny. A micro language: Generating mems resonators using a coupled form-function shape grammar. *Environment and Planning B: Planning and Design*, 27:615–626, 2000.

[5] J. T. Alander. An indexed bibliography of genetic algorithms with fuzzy logic. In W. Pedrycz, editor, *Fuzzy Evolutionary Computation*, pages 299–318. Kluwer Academic, Boston, 1997.

[6] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of The Cell*. Garland Publishing, New York, 4th edition, 2002.

[7] L. Altenberg. The evolution of evolvability in genetic programming. In K. Kinnear, editor, *Advances in Genetic Programming*, pages 47–74. MIT Press, 1994.

[8] P. Angeline and J. B. Pollack. Coevolving high-level representations. In C. Langton, editor, *Proceedings of the Third Workshop on Artificial Life*, Reading, MA, 1994. Addison-Wesley.

[9] P. J. Angeline. Morphogenic evolutionary computations: Introduction, issues and examples. In J. McDonnell, B. Reynolds, and D. Fogel, editors, *Proc. of the Fourth Annual Conf. on Evolutionary Programming*, pages 387–401. MIT Press, 1995.

[10] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–64, 1994.

[11] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proc. of the Fourth Intl. Conf. on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[12] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[13] P. Baron, A. Tuson, and R. Fisher. A voxel based representation for evolutionary shape optimisation. *Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue on Evolutionary Design*, 13(3), 1999.

[14] A. Barr. Superquadrics and angle preserving transformations. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

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

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

[17] P. J. Bentley and J. P. Wakefield. The evolution of solid object designs using genetic algorithms. In V. Rayward-Smith, editor, *Modern Heuristic Search Methods*, pages 199–215. John Wiley and Sons Inc., 1996.

[18] E. J. W. Boers and H. Kuiper. Biological metaphors and the design of modular artificial neural networks. Master's thesis, Leidea University, the Netherlands, 1992.

[19] E. J. W. Boers, H. Kuiper, B. L. M. Happel, and I.G. Sprinkhuizen-Kuyper. Designing modular artificial neural networks. In H.A. Wijshoff, editor, *Proc. of Computing Science in The Netherlands*, pages 87–96, SION, Stichting Mathematisch Centrum, 1993.

[20] E. Bonabeau, S. Guérin, D. Snyers, P. Kuntz, and G. Theraulaz. Three-dimensional architectures grown by simple 'stigmergic' agents. *BioSystems*, 56(1):13–32, 2000.

[21] J. C. Bongard and R. Pfeifer. Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Genetic and Evolutionary Computation Conference*, pages 829–836, 2001.

[22] T. Broughton, A. Tan, and P. S. Coates. The use of genetic programming in exploring 3d design worlds. In R. Junge, editor, *CAAD Futures 1997.* Kluwer Academic, 1997.

[23] I. Chen and J. Burdick. Determining task optimal robot assembly configurations. In *IEEE International Conference on Robotics and Automation.*, pages 132–137, 1995.

[24] O. Chocron and P. Bidaud. Genetic design of 3D modular manipulators. In *IEEE International Conference on Robotics and Automation*, pages 223–228, 1997.

[25] P. Coates, T. Broughton, and H. Jackson. Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 14. Morgan Kaufmann, San Francisco, 1999.

[26] D. Dasgupta and D. McGregor. Sga: A structured genetic algorithm. Technical Report IKBS-8-92, University of Strathclyde, 1992.

[27] D. Dasgupta and D. R. McGregor. Designing neural networks using the structured genetic algorithm. In I. Aleksander and J. Taylor, editors, *Artificial Neural Networks, 2*, pages 263–268, New York, 1992. Elsevier.

[28] R. Dawkins. *The Blind Watchmaker*. Harlow Longman, 1986.

[29] R. Dawkins. The evolution of evolvability. In C.G. Langton, editor, *Artificial Life*, pages 201–220. Addison Wesley, 1989.

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

[31] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries - a review. *Artificial Life*, 7(3):225–275, 2001.

[32] K. E. Drexler. Biological and nanomechanical systems. In C.G. Langton, editor, *Artificial Life*, pages 501–519. Addison Wesley, 1989.

[33] P. Eggenberger. Cell interaction for development in evolutionary robotics. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson, editors, *From Animals to Animats 4*, pages 440–448, Cambridge, MA, 1996. MIT Press Bradford Books.

[34] P. Eggenberger. Evolving morphologies of simulated 3d organisms based on differential gene expression. In P. Husbands and I. Harvey, editors, *Proc. of the 4rth European Conf. on Artificial Life*, pages 440–448, Cambridge, 1997. MIT Press.

[35] A. E. Eiben, R. Nabuurs, and I. Booij. The escher evolver: Evolution to the people. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 17, pages 425–439. Morgan Kaufmann, San Francisco, 2001.

[36] S. Farritor and S. Dubowsky. On modular design of field robotic systems. *Autonomous Robots*, 10(1):57–65, 2001.

[37] S. Farritor, S. Dubowsky, N Rutman, and J. Cole. A systems level modular design approach to field robotics. In *IEEE International Conference on Robotics and Automation*, pages 2890–2895, 1996.

[38] G. B. Fogel and D. B. Fogel. Continuous evolutionary programming: Analysis and experiments. *Journal of Cybernetics and Systems*, 26:79–90, 1995.

[39] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution*. Wiley Publishing, New York, 1966.

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

[41] J. Frazer. Reptiles. *Architectural Design*, pages 231–239, April 1974.

[42] J. Frazer. *An Evolutionary Architecture*. Architectural Association Publications, 1995.

[43] J. Frazer. Creative design and the generative evolutionary paradigm. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 9, pages 253–274. Morgan Kaufmann, San Francisco, 2001.

[44] J. Frazer and J. Connor. A conceptual seeding technique for architectural design. In *Proceedings of the International Conference on the Application of Computers in Architectural Design*, pages 425–34, Berlin, 1979. Online Conferences with AMK.

[45] K. Fredriksson. Genetic algorithms and generative encoding of neural networks for some benchmark classification problems. In *Proc. of the Third Nordic Workshop on Genetic Algorithms and their Applications*, pages 123–134. Finnish Artificial Intelligence Society, 1997.

[46] C. M. Friedrich and C. Moraga. An evolutionary method to find good building-blocks for architectures of artificial neural networks. In *Proc. of the Sixth Intl. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 951–956, Granada, Spain, 1996.

[47] P. Funes. *Evolution of Complexity in Real-World Domains*. PhD thesis, Dept. of Computer Science, Brandeis University, May 2001.

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

[49] P. Funes and J. B. Pollack. Evolutionary body building: Adaptive physical designs for robots. *Artificial Life*, 4(4):337–357, 1998.

[50] F. Gruau. Genetic synthesis of modular neural networks. In *Proc. of the Fifth International Conference on Genetic Algorithms*, pages 318–325. Morgan-Kaufmann, 1993.

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

[52] F. Gruau. Automatic definition of modular neural networks. *Adaptive Behaviour*, 3(2):151–183, 1995.

[53] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 63–70, Los Angeles, California, August 1995. In *Computer Graphics* Annual Conf. Series, 1995.

[54] M. Hemberg, U.-M. O'Reilly, and P. Nordin. GENR8: A design tool for surface generation. In *Late Breaking paper at the Genetic and Evolutionary Computation Conference*, 2001.

[55] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[56] G. S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous evolution of gaits with the sony quadruped robot. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1297–1304. Morgan Kaufmann, 1999.

[57] G. S. Hornby, H. Lipson, and J. B. Pollack. Evolution of generative design systems for modular physical robots. In *IEEE Intl. Conf. on Robotics and Automation*, pages 4146–4151, 2001.

[58] G. S. Hornby, H. Lipson, and J. B. Pollack. Generative representations for the automatic design of modular physical robots. *IEEE Transactions on Robotics and Automation*, 19(4):703–719, 2003.

[59] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation*, pages 600–607, 2001.

[60] G. S. Hornby and J. B. Pollack. Body-brain coevolution using L-systems as a generative encoding. In *Genetic and Evolutionary Computation Conference*, pages 868–875, 2001.

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

[62] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.

[63] G. S. Hornby, S. Takamura, O. Hanagata, M. Fujita, and J. Pollack. Evolution of controllers from a high-level simulator to a high dof robot. In J. Miller, editor, *Evolvable Systems: from biology to hardware; Proc. of the Third Intl. Conf.*, Lecture Notes in Computer Science; Vol. 1801, pages 80–89. Springer, 2000.

[64] C. C. Huang and A. Kusiak. Modularity in design of products and systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 28(1):66–77, 1998.

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

[66] H. Jackson. Toward a symbiotic coevolutionary approach to architecture. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 11, pages 299–313. Morgan Kaufmann, San Francisco, 2001.

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

[68] C. Jacob. Evolution programs evolved. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature PPSN-IV*, Lecture Notes in Computer Science 1141, pages 42–51, Berlin, 1996. Springer-Verlag.

[69] C. Jacob. Evolving evolution programs. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 107–115. Morgan Kaufmann, 1996.

[70] N. Jakobi. *Minimal Simulations for Evolutionary Robotics.* PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, May 1998.

[71] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, University of New Mexico, 1995.

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

[73] C. Kane and M. Schoenauer. Topological optimum design. *Control and Cybernetics*, 25(5):1059–1088, 1996.

[74] J.-O. Kim and P. K. Khosla. Design of space shuttle tile servicing robot: An application of task based kinematic design. In *IEEE International Conference on Robotics and Automation.*, pages 867–874, 1993.

[75] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

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

[77] J. Kodjabachian and J. A. Meyer. Evolution and development of modular control architectures for 1-d locomotion in six-legged animats. *Connection Science*, 10:211–237, 1998.

[78] M. Komosinski. The world of framsticks: Simulation, evolution, interaction. In *Virtual Worlds 2*, Lecture Notes in Artificial Intelligence 1834, pages 214–224. Springer-Verlag, 2000.

[79] M. Komosinski and A. Rotaru-Varga. Comparison of different genotype encodings for simulated 3d agents. *Artificial Life*, 7(4):395–418, 2001.

[80] M. Komosinski and S. Ulatowski. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In J.-D. Nicoud D. Floreano and F. Mondada, editors, *Proceedings of 5th European Conference on Artificial Life (ECAL99)*, volume 1674 of *Lecture Notes in Artificial Intelligence*, pages 261–265. Springer-Verlag, 1999.

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

[82] J. R. Koza. *Genetic Programming II : Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, Mass., 1994.

[83] P. C. Leger. *Automated Synthesis and Optimization of Robot Configurations: An Evolutionary Approach.* PhD thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1999.

[84] B. Lewin. *Genes VII.* Oxford University Press, 2000.

[85] D. S. Linden. Innovative antenna design using genetic algorithms. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 20, pages 487–510. Morgan Kaufmann, San Francisco, 2001.

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

[87] A. Lindenmayer. Adding continuous components to L-Systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 53–68. Springer-Verlag, 1974.

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

[89] S. Luke and L. Spector. Evolving graphs and networks with edge encoding: Preliminary report. In J. Koza, editor, *Late-breaking Papers of Genetic Programming 96*, pages 117–124. Stanford Bookstore, 1996.

[90] C. Mautner and R. Belew. Coupling morphology and control in an evolved robot. In Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiel, and Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 1350–1357, 1999.

[91] C. Mautner and R. Belew. Evolving robot morphology and control. In M. Sugisaka, editor, *Proc. of Articial Life and Robotics*, Oita, 1999.

[92] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[93] Z. Michalewicz, D. Dasgupta, R. G. Le Riche, and M. Schoenauer. Evolutionary algorithms for constrained engineering problems. *Computers and Industrial Engineering Journal*, 30(2):851–870, 1996.

[94] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, 1992.

[95] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural netowrks using genetic algorithms. In J. D. Schaffer, editor, *Proc. of the Third Intl. Conf. on Genetic Algorithms*, pages 379–384, San Mateo, California, 1989. Morgan Kaufmann.

[96] D. Montana and L. Davis. Training feedforward networks using genetic algorithms. In *Intl. Joint Conf. on Articial Intelligence*, pages 762–767, San Mateo, California, 1989. Morgan Kaufmann.

[97] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93 Conference Proceedings*, pages 343–350, 1993. Annual Conference Series.

[98] H. Nishino, H. Takagi, S.-B. Cho, and K. Utsumiya. A 3D modeling system for creative design. In *15th Intl. Conf. on Information Networking*, pages 479–486, Beppu, Japan, 2001.

[99] S. Nolfi and D. Parisi. Growing neural networks. Technical Report PCIA-91-15, 1991.

[100] S. Nolfi and D. Parisi. Evolving artificial neural networks that develop in time. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, editors, *European Conference on Artificial Life*, pages 353–367. Springer, 1995.

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

[102] C. Paredis, H. Brown, and P. Khosla. A rapidly deployable manipulator system. In *IEEE International Conference on Robotics and Automation.*, pages 1434–1439, 1996.

[103] J. B. Pollack, H. Lipson, G. Hornby, and P. Funes. Three generations of automatically designed robots. *Artificial Life*, 7(3):215–223, 2001.

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

[105] G. Robinson, M. El-Beltagy, and A. Keane. Optimization in mechanical design. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 6, pages 147–165. Morgan Kaufmann, San Francisco, 1999.

[106] S. Rooke. Eons of genetically evolved algorithmic images. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 13, pages 339–365. Morgan Kaufmann, San Francisco, 2001.

[107] M. Rosenman. A growth model for form generation using a hierarchical evolutionary approach. *Microcomputer in Civil Engineering*, 11:161–172, 1996.

[108] M. Rosenman and J. Gero. Evolving designs by generating useful complex gene structures. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 15, pages 345–364. Morgan Kaufmann, San Francisco, 1999.

[109] M. A. Rosenman. The generation of form using an evolutionary approach. In D. Dasgupta and Z. Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 69–85, Southampton, 1997. Springer-Verlag.

[110] G. P. Roston. *A Genetic Methodology for Configuration Design.* PhD thesis, Dept. of Mechanical Engineering, Carnegie Mellon University, December 1994.

[111] A. Rowbottom. Evolutionary art and form. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 11, pages 262–277. Morgan Kaufmann, San Francisco, 1999.

[112] D. Sankoff and J. B. Kruskal, editors. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading, Mass., 1983.

[113] M. Schoenauer. Representations for evolutionary optimization and identification in structural mechanics. In J. Periaux and G. Winter, editors, *Genetic Algorithms in Engineering and Computer Science*, chapter 22. John Wiley and Sons Ltd., 1995.

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

[115] K. Shea, J. Cagan, and S. J. Fenves. A shape annealing approach to optimal truss design with dynamic grouping of members. *Journal of Mechanical Design*, 119:388–394, September 1997.

[116] K. Sims. Artificial Evolution for Computer Graphics. *Computer Graphics*, 25(4):319–328, 1991.

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

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

[119] C. Soddu and E. Colabella. The morphogenetic design as an artificial intelligence system to support the management of design procedures through the total quality of the built-environment. In *Proc. of The Management of Information Technology for Construction. First International Conference*, Singapore, August 17-20, 1993.

[120] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7:343–351, 1980.

[121] W. A. Tackett. Greedy recombination and genetic search on the space of computer programs. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–298. Morgan Kaufmann, 1995.

[122] T. Taura and I. Nagasaka. Adative-growth-type 3d geometric representation for spatial design. *Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 13(3):171–184, 1999.

[123] T. Taura, I. Nagasaka, and A. Yamagishi. An application of evolutionary programming to shape design. *Computer-Aided Design*, 30(1):29–35, 1998.

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

[125] P. Testa, U.-M. O'Reilly, M. Kangas, and A. Kilian. Moss: Morphogenetic surface structure - a software tool for design exploration. In *Proceedings of Greenwich 2000: Digital Creativity Symposium*, 2000.

[126] S. Todd and W. Latham. *Evolutionary Art and Computers*. Academic Press, 1992.

[127] S. Todd and W. Latham. The mutation and growth of art by computers. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 9, pages 221–250. Morgan Kaufmann, San Francisco, 1999.

[128] S. J. P. Todd and W. Latham. Mutator, a subjective human interface for evolution of computer scultpures. Technical report, IBM United Kingdom Scientific Centre Report 248, 1991.

[129] K. Ulrich and K. Tung. Fundamentals of product modularity. *Issues in Design/Manufacture Integration - 1991 American Society of Mechanical Engineers, Design Engineering Division (Publication) DE*, 39:73–79, 1991.

[130] M. van de Panne and E. Fiume. Sensor-actuator Networks. In *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 335–342, 1993.

[131] J. Ventrella. Explorations in the emergence of morphology and locomotion behavior in animated characters. In R. Brooks and P. Maes, editors, *Proceedings of the Fourth Workshop on Artificial Life*, Boston, MA, 1994. MIT Press.

[132] J. Ventrella. Sexual swimmers: Emergent morphology and locomotion without a fitness function. In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson, editors, *From Animals to Animats 4*, pages 484–493, Cambridge, MA, 1996. MIT Press Bradford Books.

[133] J. Ventrella. Animated artificial life. In J.-C. Heudin, editor, *Virtual Worlds: Synthetic Universes, Digital Life, and Complexity*. Perseus Books, 1999.

[134] G. P. Wagner and L. Altenberg. Complex adaptations and the evolution of evolvability. *Evolution*, 50:967–976, 1996.

[135] M. Whitelaw. Breeding aesthetic objects: Art and artificial evolution. In P. J. Bentley and D. W. Corne, editors, *Creative Evolutionary Systems*, chapter 3, pages 129–145. Morgan Kaufmann, San Francisco, 2001.

[136] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, 14:347–361, 1990.

[137] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.