Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2003), *17*, 155–168. Printed in the USA.
Copyright © 2003 Cambridge University Press 0890-0604/03 $16.00
DOI: 10.1017/S0890060403172046

Using assembly representations to enable evolutionary design of Lego structures

MAXIM PEYSAKHOV AND WILLIAM C. REGLI

Department of Computer Science, College of Engineering, Drexel University, Philadelphia, Pennsylvania 19104, USA

(RECEIVED September 6, 2002; ACCEPTED April 11, 2003)

Abstract

This paper presents an approach to the automatic generation of electromechanical engineering designs. We apply messy genetic algorithm (GA) optimization techniques to the evolution of assemblies composed of Lego™ structures. Each design is represented as a labeled assembly graph and is evaluated based on a set of behavior and structural equations. The initial populations are generated at random, and design candidates for subsequent generations are produced by user-specified selection techniques. Crossovers are applied by using cut and splice operators at the random points of the chromosomes; random mutations are applied to modify the graph with a certain low probability. This cycle continues until a suitable design is found. The research contributions in this work include the development of a new GA encoding scheme for mechanical assemblies (Legos), as well as the creation of selection criteria for this domain. Our eventual goal is to introduce a simulation of electromechanical devices into our evaluation functions. We believe that this research creates a foundation for future work and it will apply GA techniques to the evolution of more complex and realistic electromechanical structures.

Keywords: Assembly Modeling; Computer-Aided Design; Engineering Design; Evolutionary Design; Genetic Algorithms; Lego

1. INTRODUCTION

This paper explores a graph-grammar based approach to using messy genetic algorithms (GAs) (Goldberg et al., 1989) to evolve Lego™ assemblies in an unspecified design space. Our goal is to demonstrate a new way to use evolutionary computation to solve design problems. Traditionally, GAs have been used to optimize design structures, that is, the placement of trusses or the solving of configuration or layout problems (Pollack & Funes, 1998; Koza et al., 1999; Jakiela et al., 2000). Our work continues the direction started by Simms (1994), Pollack and Funes (1997), and Coello et al. (1997) toward the evolution of the entire design. We have introduced several variations on their technique aimed at improving the representational realism of the design models that are capable of being operated on. Our empirical results show how these representations can be used to evolve real Lego assemblies, subject to parametric and mass properties.

One contribution of this work is in the representations that are used. Our experimental results show that these representations can be used to reproduce the work of Pollack and Funes while enabling the possibility of evolving much more complex designs. It is our hope that the representational ideas in the paper can be used to create more advanced evolutionary design systems.

The paper is organized as follows: after some brief background on GAs and their application to engineering design, we describe our research methodology. We then introduce, in Section 3, a unified graph-grammar based design generation tool that can evolve geometrically and structurally valid designs to solve a functionally specified set of design constraints. Section 4 provides empirical results, showing how messy GAs, coupled with a graph-grammar based representation of their assembly structure, can be used to evolve useful Lego structures. Last, Section 5 summarizes our conclusions, contributions, and ideas for future research.

Reprint requests to: William C. Regli, College of Engineering, Department of Computer Science, Drexel University, Philadelphia, PA 19104. E-mail: regli@drexel.edu

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

2. BACKGROUND

2.1. GAs and evolutionary computation

Interest in evolutionary systems for optimization problems first arose in the 1950s (Fraser, 1957). The core idea in these systems was to evolve a population of candidate solutions to a given problem, using operators inspired by natural genetics: crossover, mutation, and selection (Goldberg 1989).

GAs operate on *chromosomes* represented as collections of *genes*. Each gene is the smallest building block of the solution. Different permutations of genes create different chromosomes or individuals, which represents a solution of some quality to the given problem.

Our research uses *messy GAs* (Goldberg et al., 1989, 1990, 1993). With a messy GA, the chromosome usually does not have length constraints implying that some genes can be overspecified or underspecified. By marking some genes as dominant, we can introduce arbitration and validation modules to handle overspecification and underspecification. Messy GAs provide a kind of algorithmic flexibility that is vital to evolving engineering-like structures: because the representation length is not fixed, we can accommodate more complex assemblies with arbitrary numbers of components. In this way, a messy GA shares similar properties to genetic programming (GP; Koza et al., 1999). One focus of this work is to study how applying selection and variation operators improves on the initial randomly generated solutions.

A typical GA is shown in Figure 1. First, the initial population is generated. Second, each member of the population is evaluated. Third, one of the selection techniques is used to select candidates for the next generation. In order to form the next generation, population mutation and crossover operators are applied with fixed probabilities.

2.2. Use of GAs in engineering design

For certain problems in engineering design, GAs have been found to be very effective optimizers. They are particularly useful when the design space and the nature of the optimum solution are difficult to formalize during initial design (Bentley, 1999). Another advantage is their ability to work simultaneously with a variety of often conflicting design variables and generate sound engineering solutions.

There have been a number of significant research efforts at applying GAs to engineering design. The work of Bentley (1999) uses GAs to evolve shapes for solid objects directed by multiple fitness measures. This structural engineering problem is known as ìstructural topology optimization.î Jakiela pursued a similar approach, using GA chromosomes to represent structural topology (Jakiela et al., 1996; Jakiela & Duda, 1997). Their approach is based on converting the chromosome into topology, evaluating its fitness and structural performance, and then assigning a fitness value to it. The later work of Jakiela represents a specification-based design evaluation method and how it is applied to optimization using GAs (Jakiela et al., 2000). Jones successfully applied GAs to the evolution of antennas (Jones, 1999) using a special grammar. Each valid sentence in this grammar represents a valid antenna design, and genetic optimization is performed on sentences encoded as chromosomes. GAs and GP have also been successfully used in a number of engineering design and optimization problems as well (Koza et al., 1999). Our choice of GAs in this research is largely because they have more straightforward representational and algorithmic requirements. Use of GP for design evolution, as noted in Section 5, is an object of our current and future research study. For more general overviews on the use of GAs and GP in engineering design, interested readers are pointed to Bentley (1999), Goldberg (1989), Koza et al. (1999), and Sriram (1997).

2.3. Evolutionary design

Our work obviously relates to the ground-breaking work of Pollack and Funes (1997), who developed an encoding of Lego block structures, in particular their connectivity, in a way that enabled a GA to evolve structures that can be load bearing (i.e., bridges, cranes, etc.). They employed an elementary model of physics, which represents the connective

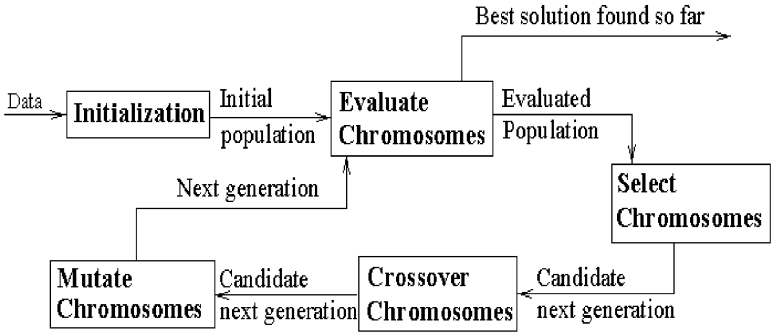


Fig. 1. A diagram of the genetic algorithm.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

strength of Lego component connections under torque and their masses, in developing their evaluation function. Essentially, their work restates the problem of designing simple load-bearing structures as an optimization problem suitable for stochastic search via a GA.

In their work they used graph-based networks of torque propagation to evaluate structures and GP, rather than a GA, to perform optimization. They also used an assembly tree to represent a Lego structure. According to Pollack and Funes (1998), their choice of representation was one of the limiting factors of their work. This point is one of the areas our research is specifically attempting to address: to develop a more sophisticated, assembly-centered representation of the design problem that can eventually be used to evolve devices with nontrivial mechanical properties.

3. TECHNICAL APPROACH

The work in this paper expands the discussion started by Pollack and Funes (1997, 1998) and Pollack et al. (1999) in the following ways:

- *Mechanisms:* Their problem formulation is limited only to design of structures created solely by Lego blocks. We have reformulated the problem in a way that includes evolution designs with mechanical properties.
- *Assembly representation:* Their design representation is based on a planar connectivity graph that enables them to propagate forces through their designs. We have created an encoding for more traditional representations of mechanical assemblies, allowing for a wider variety of Lego building blocks and for the inclusion of more sophisticated connectivity among primitives (i.e., motion properties and constraints).
- *Lego Shape Grammar:* We have created a shape grammar for describing valid configurations of Lego assemblies. The grammar is used for validity checking of assembly configurations.
- *Lego Solid Models:* Our Lego models are realized as 3-dimensional (3-D) solid models, allowing us to more accurately model the physics of the design, as well as reason about kinematic properties (i.e., interference, collision detection, etc.).

3.1. Why study Legos?

We selected Legos because they represent a sufficiently complex, multidisciplinary design domain that includes a wide variety of realistic engineering constraints. Further, the domain is sufficiently discrete as to be tractable. Solutions to problems in the Lego domain will have great practical value. First, the growing popularity of Lego robots and Lego-based education and competitions (Kitano et al., 1997) makes this domain an ideal testbed for ideas and software tools for design generation or optimization. Second, the generation of Lego assemblies is closely related to the generation of real engineering artifacts: concepts that work in the Lego domain may reveal principles that can be extended to similar problems in more complex industrial domains.

3.2. Problem formulation

In order to evolve Lego assemblies, we modified the basic GA evolutionary loop and added several modules specific to the domain. First, we include a *Lego Element Repository*, describing the types, dimensions, and physical appearance of available Lego elements. Second, we developed a separate module to perform design validation and handle overspecified and underspecified chromosomes. This module uses the Lego graph grammar rules to parse and validate the structure. The third significant customization was a graphical output and visualization subsystem. This subsystem allows us to visualize Lego structures as they are evolving. The diagram of our GA for Legos is shown on Figure 2.

3.3. Representing Lego components and physical assemblies

We used assembly graphs that capture contact and feature-based connectivity (i.e., joints, parameters, etc.) to represent Lego assemblies. Representing Lego designs as a mechanical assembly graph has a number of potential advantages over the assembly tree approach suggested in earlier research (Pollack & Funes, 1997). A labeled assembly graph is more expressive and can represent greater variety of the Lego assemblies, including kinematic mechanisms and static structures (Schmidt et al., 1999). The nodes of the graph represent different Lego elements, and the edges of the graph represent connections between elements. We developed a notation for describing valid Lego assemblies based on a graph grammar, similar to those described by Schmidt et al. (1996) and Schmidt and Cagan (1998). The idea of utilizing graph grammars for design representation was also presented by Rosen et al. (1994). In our system, the graph grammar defines assembly validity and enables us to describe legal combinations of the nodes and edges precisely and unambiguously.

Each Lego structure is represented by assembly graph G. Assembly graph G is a directed labeled graph with a nonempty set $N(G)$ of nodes n, representing Lego elements, and set $E(G)$ of edges e, representing connections (i.e., mating, contact) and relations (i.e., joints) between those elements (Chase, 1996). Simple Lego structure and corresponding assembly graph are shown in Figure 3. The node label contains the type and the parameters of the element. For now, our program can operate only with three types of Lego elements, namely Beam, Brick, and Plate. We will combine these three types under the category Block.

The number and nature of parameters specified in the label depends on the type of element. For Beam, Brick, and

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

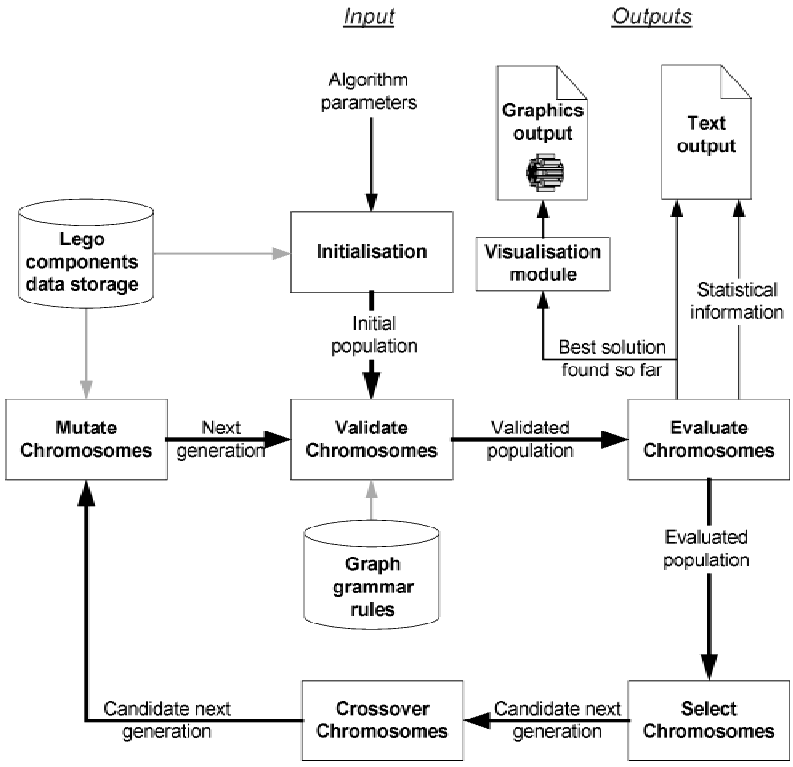


Fig. 2. The genetic algorithm specialized for evolving Lego assemblies.

Plate, these parameters are the number of pegs on the element in X and Y dimensions. The same parameters also define the orientation of the block in space. In this way the beam $[6, 1]$ is aligned along the x axis and beam $[1, 6]$ along y. We have created a labeling scheme for the elements of type Axle, Wheel, and Gear, although these are not yet part of our empirical studies.

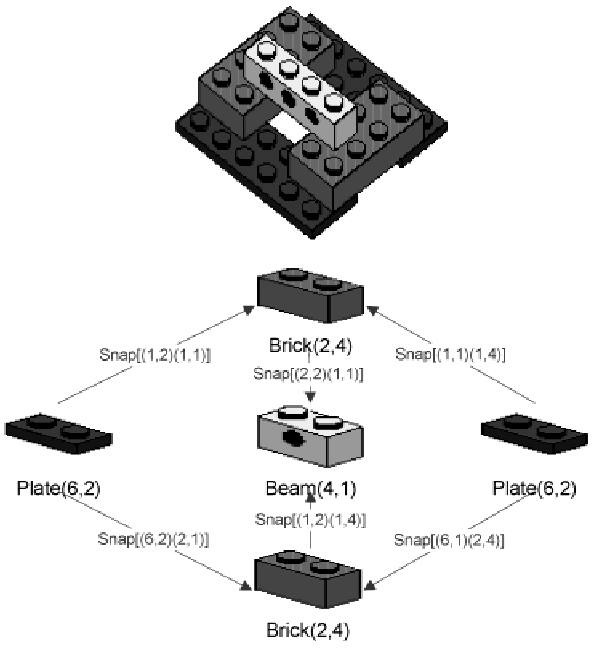


Fig. 3. An example Lego structure with an assembly graph.

The edge label contains the parameters defining the connection. An edge can be directed or undirected, depending on the type of the connection. For now our program can operate only with one type of Lego connection, a Snap Fit, because this is the only possible connection between Brick, Beam, and Plate elements. Snaps are directed edges with arrows pointing from the Block, which provides pegs to the Block that serve as connection surfaces. The Edge label contains the connection type Snap and a Peg-Pair that is used to define the Pair of corresponding pegs in the connection [(Poz$X$1, Poz$Y$1), (Poz$X$2, Poz$Y$2)]. (Poz$X$1, Poz$Y$1) defines the peg on the Block, which provides pegs, and (Poz$X$2, Poz$Y$2) defines the peg on the Block, which provides the connection surface. This means that the peg on the block that Snap is pointing to is always defined second in the Peg-Pair. Such a pair uniquely defines the set of mating features for both blocks, given the implicit orientation of the elements. Examples of Lego nodes and edges are shown in Figure 4.

3.4. Lego shape grammar

A parameterized context-sensitive directional graph grammar was designed to handle 3-D structures assembled from

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

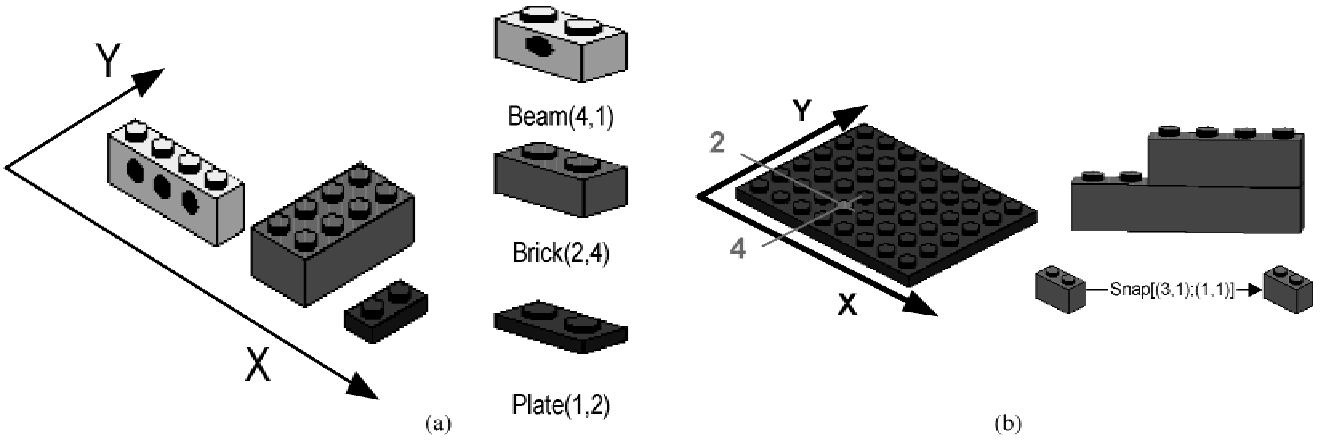


Fig. 4. An example of the nodes and edges. (a) Lego blocks (left) with labeled nodes (right) and (b) Peg coordinates and Snap connections.

Lego blocks, axles, and wheels. The grammar vocabulary is[1]

{•Mechanism, •Module, |Connect, •Block, •Element, •Disk, •Pole, PegPair, ↑Snap, |Insert, |Tinsert, |Gtrans, •Beam, •Brick, •Plate, •Wheel, •Gear, •Axle, Poz*X*, Poz*Y*, Size*X*, Size*Y*, Len, Diam, Teeth, Hole (,), [,], ;} .

The starting word of the grammar is {Mechanism}. The terminal words of the grammar are:

{↑Snap, |Insert, |Tinsert, |Gtrans, •Beam, •Brick, •Plate, •Battery, •Motor, •Wheel, •Gear, •Axle, Size*X*, Size*Y*, Len, Poz*X*, Poz*Y*, Diam, Teeth, Hole (,), [,], ;} .

Terminal symbols ( )[ ],; are used to make sentences easier to read, so most often they will not be shown in our subsequent derivations and Lego syntax trees.

Terminals Poz*X*, Poz*Y*, Size*X*, Size*Y*, Len, Diam, Hole, and Teeth are parameters. The meanings of the parameters Size*X*, Size*Y*, Poz*X*, and Poz*Y*, which are used to define Lego blocks and their connections, was described in the previous paragraph.

These sets can be modified according to the Lego brick standards. Len specifies the number of pegs on the Lego Beam or the length of an Axle measured in the peg sizes. Terminal Hole defines the sequential number of the hole in the Beam in to which Axle is inserted. Diam reflects the diameter of the wheel. Teeth represent the number of teeth on the Lego Gear. This number also uniquely defines the diameter of the Gear. All of the sizes defined in terms of Lego units, but there is one to one correspondence between Lego units and millimeter measurements of each element as demonstrated in Figure 5.

[1]For notational purposes, "•" corresponds to nodes in the graph grammar, "|" to an undirected edge and "↑" to a directed edge.

$$\text{Poz}X \in \{1 \,..\, \text{Size}X\},$$

$$\text{Poz}Y \in \{1 \,..\, \text{Size}Y\}$$

$$\text{Size}X \in \{1, 2, 4, 6, 8, 10, 12, 16\}$$

$$\text{Size}Y \in \{1, 2, 4, 6, 8, 10, 12, 16\}$$

$$\text{Len} \in \{1, 2, 4, 6, 8, 10, 12, 16\}$$

$$\text{Hole} \in \{1 \,..\, (\text{Len} - 1)\}$$

$$\text{Diam} \in \{17, 30, 43\}$$

$$\text{Teeth} \in \{8, 16, 24, 40\}$$

•Mechanism, •Module, •Element, •Block, •Disk, •Pole, •Beam, •Brick, •Plate, •Battery, •Motor, •Wheel, •Gear and •Axle are the nodes of the graph grammar.

$$\forall\, x \in N(G), x \in \{\text{•Block, •Disk, •Pole, •Beam, •Brick, •Plate, •Battery, •Motor, •Wheel, •Gear, •Axle}\}$$

A Module represents a number of Elements connected together. An Element represents a single Lego piece. The grammar includes three categories of Lego elements, namely Block, Disk, and Pole.

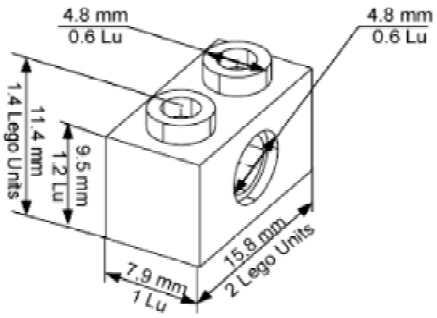


Fig. 5. A Lego block with both official Lego units and metric dimensions.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

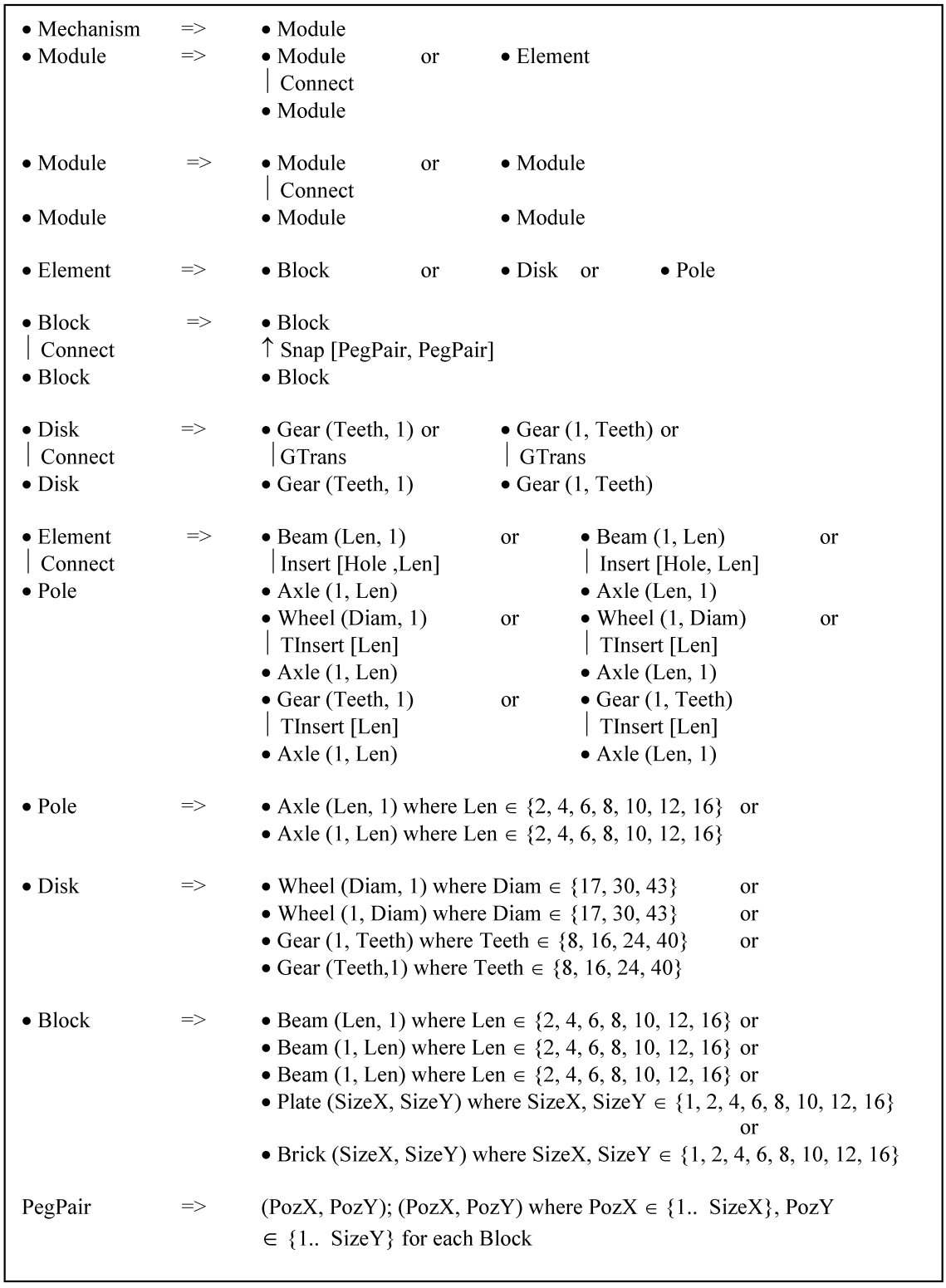


Fig. 6. Examples of Lego grammar rules.

|Connect, ↑Snap, |Insert, |TInsert, and |GTrans are the edges of the graph grammar.

$$\forall\, y \in E(G), \quad y \in \{\uparrow \text{Snap}, |\text{Insert}, |\text{Tinsert}, |\text{GTrans}\}$$

Based on this formulation, one could build another level of abstraction that represents Lego mechanisms as a sentence in a language of Lego assemblies rather than as a graph, making it easy to validate the assembly against

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

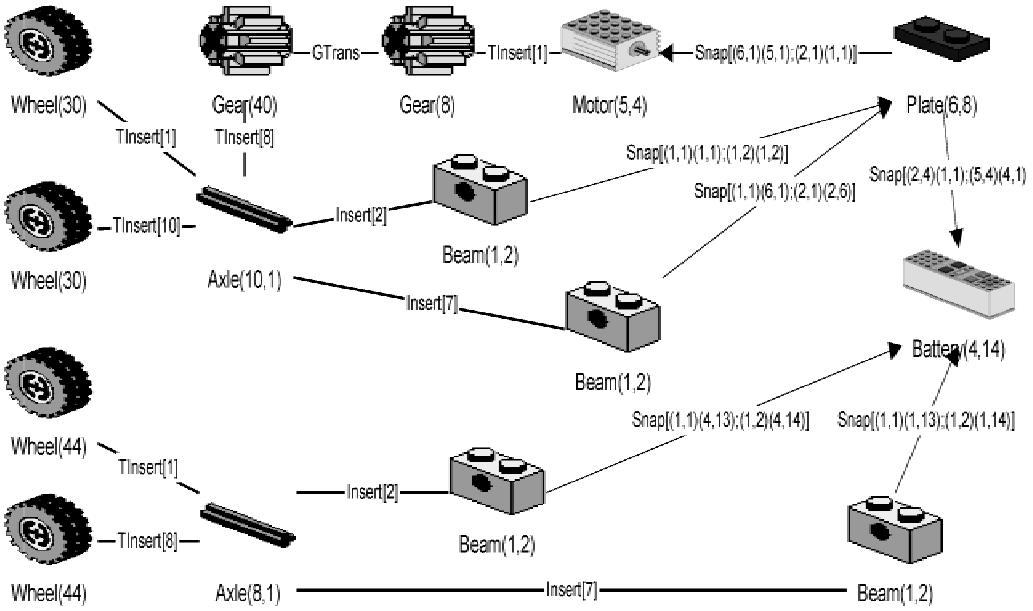


Fig. 7. An assembly graph for the Lego car.

grammar rules (Jones, 1999). This is a problem open for future work.

A portion of our Lego graph grammar is shown in Figure 6. We have developed the specifications for representation of wheels, gears, and axles and their connections, an example of which is the assembly graph shown in Figure 7 for the Lego car in Figure 8.

This Lego language is useful for classifying the Lego blocks and their connections. In the following sections, we use the grammar to validate designs produced by evolutionary computation. It should be noted that the Lego grammar itself could form a basis for design generation and evaluation, as has been done in previous research (Heisserman & Woodbury, 1993; Schmidt et al., 1996). Although this is

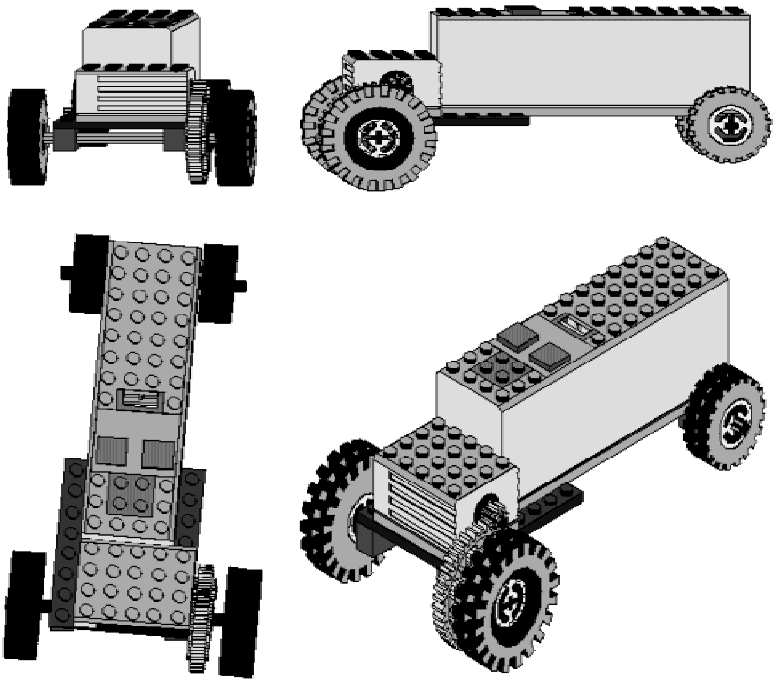


Fig. 8. An example of the Lego car.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

0	Plate(6,2)
1	Plate(6,2)
2	Brick(2,4)
3	Brick(2,4)
4	Beam(4,1)

0>2	Snap[(1,2)(1,1)]
0>3	Snap[(6,2)(2,1)]
1>2	Snap[(1,1)(1,4)]
1>3	Snap[(6,1)(2,4)]
2>4	Snap[(2,2)(1,1)]
3>4	Snap[(1,2)(1,4)]

Fig. 9. The chromosome of the example structure.

theoretically, possible, the complexity of the Lego domain makes the size of the search space for a direct grammar-based approach computationally intractable. Subsequent sections will show that a GA is an effective means of navigating this search space.

3.5. Encoding scheme for GA

In order to perform a GA-based optimization on the population of Lego assembly graphs, we first have to define a way to encode the assembly graph as a chromosome. Our chromosome is represented by a combination of two data structures: an array containing all nodes $N(G)$ and the adjacency hash table containing all edges $E(G)$ with corresponding string keys as demonstrated on Figure 9. This array is called the *genome*, and individual elements of the array are called *genes*. The genome defines what Lego elements compose the structure and how they are connected. The key value of the hash table is used to represent the γ function of the graph G and defines the position and direction of an edge. For example, the key $1 > 3$ is equivalent to the key $3 < 1$ and means that the edge is located between nodes 1 and 3 and directed to node number 3. A hash table defines the way Lego elements will be connected together.

3.6. Genetic operations on assembly graphs

3.6.1. Initialization

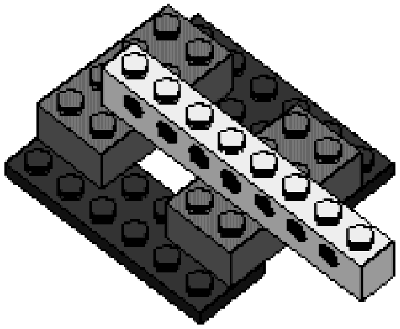
The initial population is generated at random. First, 10 nodes of random Lego types are generated with random dimensions. Then, 9–13 edges (e.g. connections) are generated and placed at random subject to the constraint that the resulting structure must be physically feasible. In the future we will look into creating and applying initialization rules to promote exploration of specific areas of the fitness landscape. As an example, one of the rules can be: all chromosomes must have at least one Lego element of the specific type. At present, we rely on a purely randomly generated initial population.

3.6.2. Mutation

In order to provide the balance between the exploration and exploitation, the mutation operator is applied with a constant, low probability. The mutation operator works on the genome array of the mutated chromosome: in this context, after a gene is selected for mutation, it is simply replaced with a Lego element of the same type and random size as shown on Figure 10. Some edges might become invalid after element mutated; in these cases the edges are reinitialized at random. Obviously, mutation has limited ability to change the structure on the graph and works mostly on the nodes themselves. An area for future study is how to introduce a mutation that alters edges and the small subgraphs.

3.6.3. Crossover

As in the majority of messy GAs, crossover is performed with the help of two genetic operators: *cut* and *splice*. When two chromosomes are chosen for crossover, the cut operator applied to both of them at independently selected random points. Then, the tail parts of the chromosomes are spliced with the head parts of the other chromosome. Because selection of the crossover point is independent for each chromosome, it is obvious that the chromosome length can vary during the evolution process to allow for the evo-



0	Plate(6,2)
1	Plate(6,2)
2	Brick(2,4)
3	Brick(2,4)
4	Beam(8,1)

0>2	Snap[(1,2)(1,1)]
0>3	Snap[(6,2)(2,1)]
1>2	Snap[(1,1)(1,4)]
1>3	Snap[(6,1)(2,4)]
2>4	Snap[(2,2)(1,1)]
3>4	Snap[(1,2)(1,4)]

Fig. 10. The sample structure with a mutated beam and its genome.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

lution of assemblies with different number of elements and for the assembly structure to grow in complexity and size. In order to cut the chromosome, a random point P is selected between 1 and $N-1$, where N is the number of genes in the genome. Then, all genes with a number less than P are considered the head segment and all genes with a number greater or equal to P are considered a tail segment.

All edges between nodes of two different segments are deleted, but all edges within the segments are preserved. In order to splice two segments together the head and tail segments are placed into one chromosome, which produces a disjoint graph. Then the subgraphs are joined with a small number of randomly generated edges. Figure 11 shows the results of cut and splice operators applied to the sample Lego structure.

3.6.4. Evaluation function

Each Lego structure has a number of attributes, including weight, number of nodes, and size in each dimension. These parameters are used by the evaluation function to calculate fitness of the structure. Our eventual goal is to introduce a simulation of electromechanical devices into our evaluation function and be able to evaluate the ability of an assembly relative to its performance and function. Generally, evaluation function was created according to the following form:

$$\frac{1+\sum P_i(a_i)}{1+\sum P_i(b_i)+\sum P_i(|c_i-t_i|)}$$

where P_i is the weight function, which represents the importance of evaluated parameter. For the most critical parameters it was set equal to the value of the parameter itself. For less important ones it was set to be the square root of the parameter, and for the least important ones it was set to be the square root of the square root of the parameter.

$$P_i = x_i, \quad P_i = x_i^{1/2}, \quad P_i = x_i^{1/4}.$$

In addition, a_i denotes the properties the user wants to maximize unconditionally; reliability can serve as example for such a parameter; b_i denotes the properties the user wants to minimize, such as the manufacturing cost; and c_i are properties we want to bring as close as possible to the specific constant t_i. Sizes in the x–y–z dimension can be a good example of this type of properties.

3.6.5. Handling overspecified/underspecified chromosomes

As in most messy GAs, there may be *overspecified* or *underspecified* chromosomes generated during the evolution process (Goldberg, 1989). That necessitates the step of validation against the Lego grammar described in Section 3.4 and possibly repair of the individuals. Underspecification means that not all of the required information is present in the chromosome. Most often it results in the disjoint assembly graph. In cases like this, the nodes of the subgraph containing the zeroth node are selected to be dominant. The submissive subgraph is not deleted from the chromosome, but is ignored in most calculations. Figure 11a can demonstrate an example of the underspecified chromosome (i.e., a disjoint Lego assembly graph). On the other hand, an overspecified chromosome has more than one value for the same gene. In Lego structures it either results in the blocks sharing the same physical space or being connected by the set of edges, which imply two different locations for the same node. In the first case, the node that was assigned the location first is marked as dominant and the node that was assigned the location second is marked as submissive and ignored in most calculations. In the case where different edges imply different locations for the same block edge, the one that traversed first is given priority and other edges are marked as submissive and ignored. An example of the overspecified chromosome is shown on Figure 12.

4. EMPIRICAL RESULTS

4.1. Description of prototype system

Our system was extended from the *sGA* system originally created by Hartley (1998) and written in the Java programming language. Java3D and VRML97 were used in order to

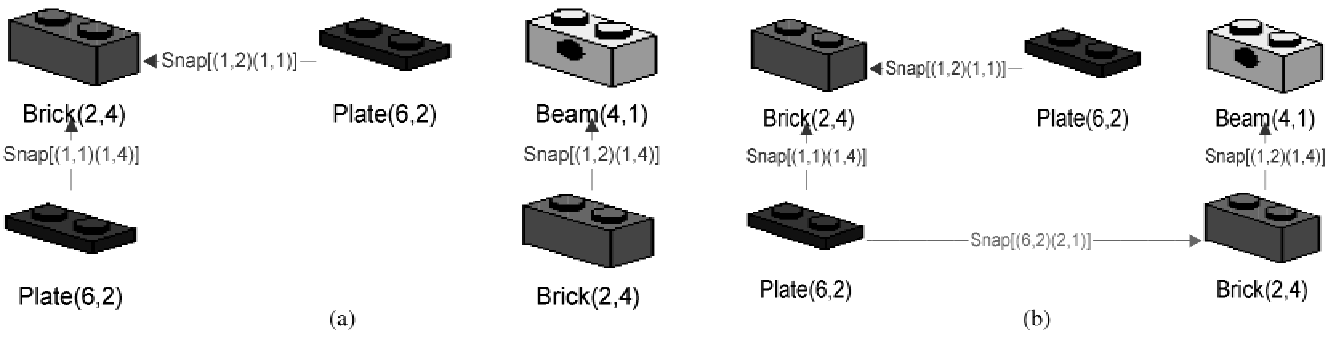


Fig. 11. A demonstration of cut and splice operators. (a) The sample structure after a cut operator is applied at point 3 and (b) head and tail subgraphs spliced with one random edge.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

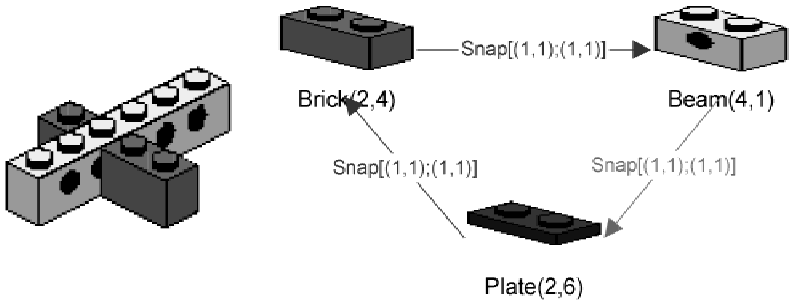


Fig. 12. An example of the blocks sharing the same physical space (left) and infeasible connection edges (right).

create a visualizer to monitor Lego structures as they evolve. The system supports one-point crossover. We are planning to introduce a uniform and *N*-point crossover in the future. The user can choose from proportional, rank, universal stochastic sampling, sexual, sigma scaling, and Boltzmann selection techniques. Other parameters to the system include mutation and crossover rates and population size. Although the current system can only handle static Lego structures composed of block-type elements, it is our belief that the general approach can be applied to much more elaborate kinematic mechanisms.

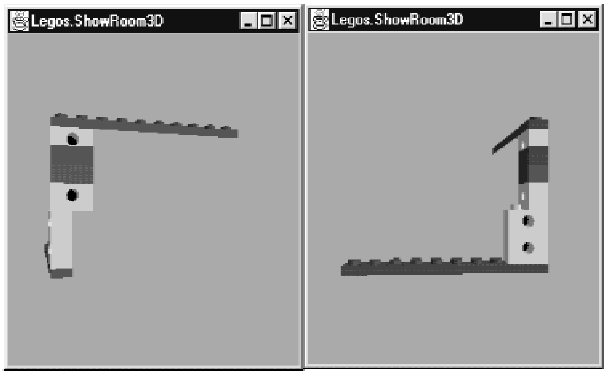
4.2. Experiments

4.2.1. Evolving 10 × 10 × 10 structure

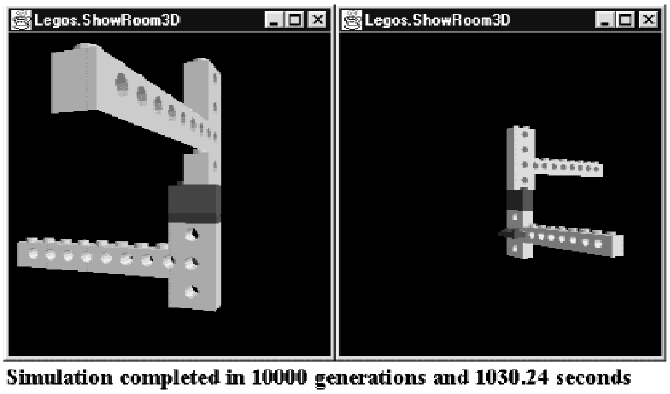
Figure 13a demonstrates the result of 1000 generations of evolution of the static Lego structure with predefined geometric parameters. In this case the goal was to evolve a structure with the size of 10 Lego units in each *x*–*y*–*z* dimension with the minimal weight. In this experiment the mutation and crossover rates were 0.01 and 0.7, respectively, and we employed a rank selection strategy and elitism on the population of 100 members. The resulting structure was discovered at generation 895 and had the sizes $10 \times 10 \times 6.8$, which is sufficiently close to the desired result. Further, we note that this is one of the lightest possible structures that satisfy these parameters that can be created from the set of elements given. Another run of the similar experiment is shown in the Figure 13b. This structure as discovered by GA in generation 3367; it is a little bit heavier, but it has a perfect $10 \times 10 \times 10$ size.

4.2.2. Evolving pillarlike structure

Another line of experiments is shown in Figure 14, where we were evolving a pillarlike structure. The goal was to



(a)



(b)

Fig. 13. An example of the light structure with the desired size of $10 \times 10 \times 10$: (a) $10 \times 10 \times 6.8$ and (b) $10 \times 10 \times 10$.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

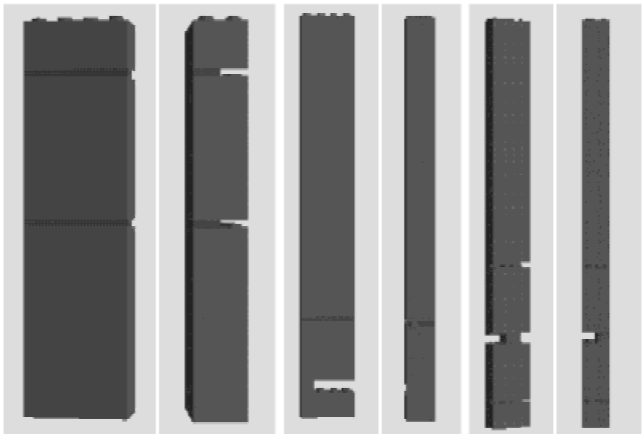


Fig. 14. An example of the pillar structure.

make an assembly with 4×2 base in the x–y dimension and 20, 40, and 60 length in the z dimension. A second constraint we specified was density: the pillar should have as few holes as possible. We used the same parameters as in the first experiment and ran the simulation for various time intervals. On average, a solution was discovered within 5000 generations. All the structures have the desired size and very few defects.

4.2.3. Evolving wall-like structure

The goal of this cycle of experiments was to evolve a wall-like structure. The desired structure should have width of 10 units, height of 20 units, and depth of 1 Lego unit. Maintaining the 1 unit depth had higher priority then reaching the desired height and width. The third goal was to increase the mass of the structure while keeping it confined to these dimension. This resulted in a dense, uniform structure with few or no defects. The structure is shown in Figure 15, and it satisfies the maximum density requirement. This structure was evolved in 91,506 generations.

4.2.4. Evolving a staircase structure

In these experiments, we attempted to evolve a staircase and succeeded in creating staircases of 3, 4, 5, 6, 7, and 10 steps (10 was the maximum attempted). The staircases of 10 and 7 steps are shown in Figure 16. In order to define the shape of the staircase, we used the following simple constraint: the structure width had to be equal to the number of steps plus one. Getting this dimension right would give the individual a higher score then getting any other dimension right. The desired height was set to a number of steps times 0.4 (the height of the Lego plate) and the depth to 1 Lego unit. We also targeted the number of bricks used in the structure to be equal to the number of steps. In order to achieve these results we had to limit our search space by providing only 1×2 Lego plates for the algorithm to use. The structures of 10 and 7 steps were evolved in 568 and 51,573 generations, respectively.

4.3. Discussion of results

Although the initial results are promising, this research is in its early stages. We are planning to introduce more types of Lego elements and connections, specifically for encoding elements of the type Wheel, Gear, and Axle, as well as connections between them. Other types of Lego elements, such as Motors, Batteries, and their connections, are still to be formally defined and implemented. We are also planning to develop more a realistic representation of the physics,



Fig. 15. An example of the wall structure.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

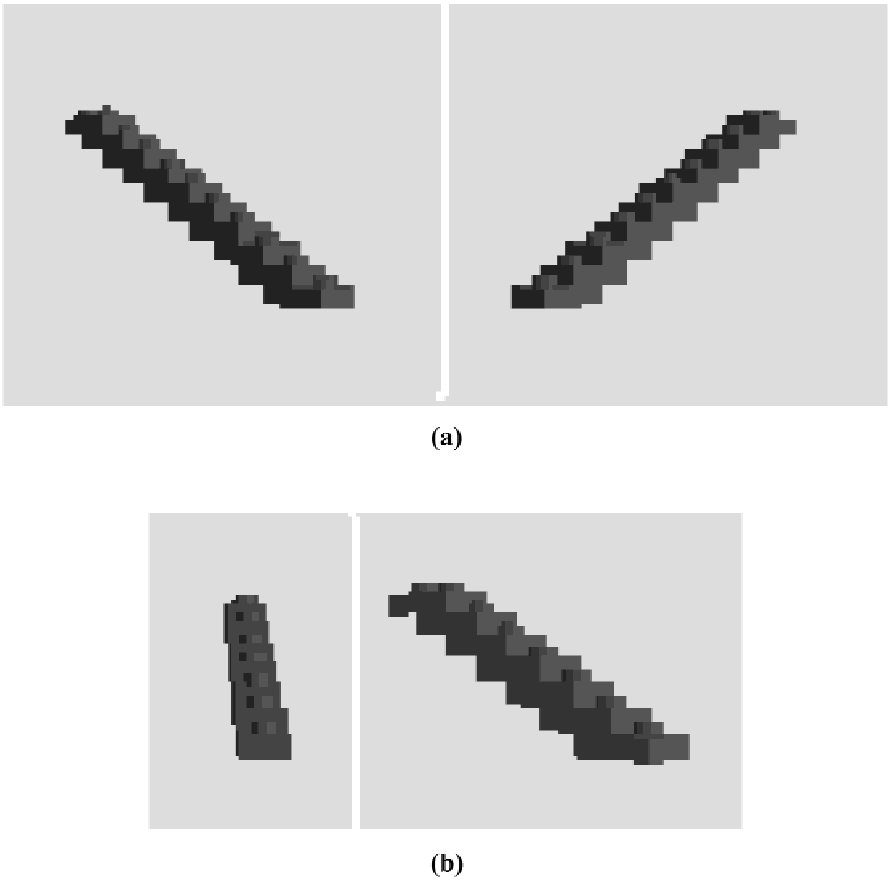


Fig. 16. A staircase example of (a) a 10-step structure and (b) a 7-step structure.

based on solid models, to help us better evaluate the structures. All of these enhancements will help us generate Lego structures and mechanisms more fitted to perform a specific task.

In our experiments we noticed that the algorithm was much better at evolving structures that can be represented as a sparse graph, linked list, or tree than it was as evolving structures that have to be represented as a highly connected graph. We think that this is due to the highly disruptive nature of the cut operator as it applies to the assembly graphs. One of the most important directions of future research would be the development of *cellular encoding* (Gen & Cheng, 1999) or *edge encoding* (Luke & Spector, 1996) for the problem. Cellular encoding and edge encoding are ways to evolve the instructions for graph development, rather than evolve the graph itself. Using these encodings should lead to better continuity of the phenotype over the generations and also make development of symmetric objects more feasible (a highly desired quality for most mechanisms). These encodings would be based on the Lego grammar, thus simplifying the process of validation and development into an actual assembly graph.

There are also a number of improvements that can be made to the GAs. For example, creating a mutation operator that would alter small subgraphs will help the algorithm to explore areas of the fitness landscape neighboring to the solution. The GA could also be augmented with a deterministic search that would run periodically for a limited time, allowing the GA to discover neighboring solutions and, hopefully, increase the time efficiency of the algorithm.

Presently, the cut operator is applied at random points, which often separate elements that should work together into two different subgraphs. We are planning to introduce a notion of clusters—highly interconnected subgraphs loosely coupled together—that would correspond to the separate modules or "organs." The GA can then be adjusted to promote crossover on the cluster boundaries and demote crossover, which breaks up a cluster.

We also noticed that the use of elitism usually greatly speeds up the search. One of the experiments we are proposing is to analyze the common features among best performing chromosomes and generate new elite chromosomes based on these features rather then simply copy them from the previous generation. Another improvement, which could also significantly speed up the search, is to use guided or seeded initialization. This would mean introducing absolute and probabilistic rules and applying them during the generation of the initial population or injecting prebuilt mechanisms into the initial population.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

5. CONCLUSIONS, CONTRIBUTIONS, AND FUTURE WORK

This paper introduced our approach, prototype system, and initial experimental results toward the evolution of Lego structures. The main research contributions described in this paper are the development of a graph-grammar based representation scheme for Lego assemblies and its encoding as an assembly graph for manipulation with and evolution by messy GAs. This graph-based approach is unlike other systems for Lego design evolution (Pollack & Funes, 1998), and we believe that this assembly graph-grammar based representation scheme is one of the most general ways of representing the assembly. In this way it provides a more flexible means to represent a wide variety of mechanisms for use with GAs. Although the present system uses the grammar sparingly and all of the rules are hard-coded in the implementation, we believe that the framework can be extended to perform parsing and evolution of general Lego sentences.

Our work contributes to several areas of computer science and engineering. Fundamentally, we believe that this work demonstrates how to integrate more sophisticated and realistic engineering models for use in evolutionary computation. From the standpoint of engineering design, we believe that our work is part of the ongoing discussion about the utility and limits of evolutionary computation for engineering design problems. Although our empirical results do not show automated generation of full-scale electro-mechanical devices, we believe that our work, coupled with higher fidelity models of mechanical kinematics, could indeed lead to the creation of evolutionary design systems that could scale to the general domain of Lego devices.

Many areas for future work have been touched on in the paper. It is our hope that this research opens opportunities for other researchers, especially those with domain-specific design problems other than Legos, to adopt and extend the framework presented in this paper. Our future work will include continuing to improve the design representation and incorporating models of function and behavior, in addition to physical properties and mechanical constraints. We have long-term hopes of creating a coherent model of design semantics, one that might be used to enable truly knowledge-intensive design applications.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation, Knowledge and Distributed Intelligence in the Information Age (KDI) Initiative Grant CISE/IIS-9873005 and Career Award CISE/IRIS-9733545. Additional support was provided by the Office of Naval Research (Grant N00014-01-1-0618).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the other supporting government and corporate organizations.

REFERENCES

Bentley, P. (1999). *Evolutionary Design by Computers*. San Francisco, CA: Morgan Kaufmann.

Chase, S. (1996). Representing designs with logic formulations of spatial relations. In *Workshop Notes, Visual Reasoning and Interaction in Design, Fourth Int. Conf. Artificial Intelligence in Design*.

Coello, C., Christiansen, A., & Aguirre, A. (1997). Automated design of combinational logic circuits using genetic algorithms. *Proc. Int. Conf. Artificial Neural Nets and Genetic Algorithms, ICANNGA'97*, pp. 335–338.

Fraser, A.S. (1957). Simulation of genetic systems by automatic digital computers. *Australian Journal of Biological Sciences*, *10*, 484–491.

Gen, M., & Cheng, R. (1999). *Genetic Algorithms and Engineering Optimization*. Hoboken, NJ: Wiley.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston: Addison–Wesley.

Goldberg, D., Deb, K., Kargupta, H., & Harik, G. (1993) IlliGAL Report No.93004. *Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms*. Urbana, IL: University of Illinois.

Goldberg, D., Deb, K., & Korb, B. (1990). Messy genetic algorithms revised: Studies in mixed size and scale. *Complex Systems*, *4*, 415–444.

Goldberg, D., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation analysis and first results. *Complex Systems*, *3*, 493–530.

Hartley, S. (1998). *Concurrent Programming: The Java Programming Language*. Oxford, UK: Oxford University Press.

Heisserman, J., & Woodbury, R. (1993). Generating languages of solid models. *Proc. Second Symp. on Solid Modeling and Applications '93*, pp. 103–112.

Jakiela, M., Chapman, C., Duda, J., Adewuya, A., & Saitou, K. (2000). Continuum structural topology design with genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, *186(2–4)*, 339–356.

Jakiela, M., & Duda, J. (1997). Generation and classification of structural topologies with genetic algorithm speciation. *Journal of Mechanical Design*, *119(1)*, 127–130.

Jakiela, M., Wallace, D., & Flowers, W.C. (1996). Design search under probabilistic specifications using genetic algorithms. *Computer-Aided Design*, *28(5)*, 405–421.

Jones, E. (1999). *Genetic design of antennas and electronic circuits*. PhD Thesis. Durham, NC: Duke University.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., & Osawa, E. (1997). RobоCup: The robot world cup initiative. *Proc. First Int. Conf. Autonomous Agents*, pp. 340–347.

Koza, J., Bennett, F., III, Bennett, F., Andre, D., & Keane, M. (1999). *Genetic Programming III: Automatic Programming and Automatic Circuit Synthesis*. San Francisco, CA: Morgan Kaufmann.

Luke, S., & Spector, L. (1996). Evolving graphs and networks with edge encoding: Preliminary report. *Late Breaking Papers at the Genetic Programming 1996 Conf.*, pp. 117–124.

Pollack, J., & Funes, P. (1997). Computer evolution of buildable objects. *Fourth European Conf. Artificial Life*, pp. 358–367.

Pollack, J., & Funes, P. (1998). Evolutionary body building: Adaptive physical designs for robots. *Artificial Life*, *4*, 337–357.

Pollack, J., Watson, R., & Ficici, S. (1999). Embodied evolution: Embodying an evolutionary algorithm in a population of robots. In *1999 Congress on Evolutionary Computation* (Angeline, P., Michalewicz, Z., Schoenauer, M., Yao, X., & Zalzala, A., Eds.), pp. 335–342. Piscataway, NJ: IEEE.

Rosen, D., Dixon, J., & Finger, S. (1994). Conversion of feature-based design representations using graph grammar parsing. *ASME Journal of Mechanical Design*, *116(3)*, 785–792.

Schmidt, L., & Cagan, J. (1998). Optimal configuration design: An integrated approach using grammar. *Transactions of the ASME, Journal of Mechanical Design*, *120(1)*, 2–9.

Schmidt, L., Shetty, H., & Chase, S. (1996). A graph grammar approach for structure synthesis of Mechanisms. *Journal of Mechanical Design*, *122(4)*, 371–376.

Schmidt, L., Shi, H., & Kerkar, S. (1999). The "Generation gap": A CSP approach linking function to form grammar generation. *Proc. DETC99, DETC99/DTM-048*.

Simms, K. (1994). Evolving virtual creatures. *Artificial Life*, *4*, 28–39.

Sriram, R.D. (1997). *Intelligent systems for engineering: A knowledge-based approach*. Berlin: Springer Verlag.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press

Maxim Peysakhov is a PhD student in the Department of Computer Science at Drexel University. He received an MS in software engineering in 2000 and a BS in computer science with a minor in mathematics in 1998, both from Drexel University. Maxim's research concentrates on algorithms inspired by nature. He has published work in the areas of evolution-based techniques, automatic design generation, and software agents.

William Regli is an Associate Professor in the Department of Computer Science at Drexel University. He holds a courtesy appointment in the Department of Mechanical Engineering and Mechanics and is Director of Drexel's Geometric and Intelligent Computing Laboratory. Dr. Regli received the PhD in computer science in 1995 from the University of Maryland at College Park and a BS (Cum Laude) in mathematics and computer science in 1989 from Saint Joseph's University in Philadelphia. He is the recipient of a 1998 National Science Foundation Career Award, the University of Maryland Institute for Systems Research Outstanding Graduate Student Award (1994–1995), a NIST Special Service Award (1995), a General Electric Corporation Teaching Incentive Grant (1994–1995), among other awards. He is a member of ACM, IEEE Computer Society, AAAI, and Sigma Xi. Dr. Regli has authored or coauthored more than 100 technical publications.

https://doi.org/10.1017/S0890060403172046 Published online by Cambridge University Press