Using Graph Grammars and Genetic Algorithms to Represent and Evolve Lego Assemblies

Maxim Peysakhov
GIC Lab, MCS Department
Korman Computing Center
Drexel University
Philadelphia, PA 19104

Vlada Galinskaya
GIC Lab, MCS Department
Korman Computing Center
Drexel University
Philadelphia, PA 19104

William C. Regli
GIC Lab, MCS Department
Korman Computing Center
Drexel University
Philadelphia, PA 19104

Abstract

This research presents an approach to the automatic generation of electromechanical engineering designs. Our intent is to apply the Messy Genetic Algorithm optimization techniques to the evolution of assemblies composed of the Lego structures. Each design is represented as a labeled assembly graph. Designs are evaluated based on a set of behavior and structural equations, which we are trying to optimize. Our eventual goal is to introduce a simulation of electromechanical devices into our evaluation functions. The initial populations are generated at random. The design candidates for subsequent generations are produced by user specified selection technique. 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 will continue 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. 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

Genetic Algorithms, Assembly Modeling, Computer-Aided Design, Engineering Design, Lego.

1 INTRODUCTION

This paper explores a graph-grammar-based approach to use Genetic Algorithms (GAs) to evolve Lego assemblies in an unknown design space.

After some brief background on Genetic Algorithms and their application to Engineering Design, we describe our initial research goal: to create 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. We provide a description of the representation scheme for Lego assemblies and describe how a GA can be applied to evolve Lego structures.

Our algorithm combines the knowledge of physical properties of Lego components and evolutionary process to create more complex mechanisms. Mechanisms are then evaluated to determine how well their structure satisfies the initial design goals and how well the design conforms to the desired attributes and performance functions.

2 BACKGROUND

2.1 APPLICATIONS OF GAS IN ENGINEERING DESIGN.

For certain problems of engineering design, genetic algorithms have been found to be very effective optimizers. GAs are particularly useful when the design space and the nature of the optimum solution is difficult to formalize during the initial design [7]. Another advantage is their ability to work simultaneously with a variety of design variables.

There have been a number of significant research efforts at applying GAs to engineering design. The work of Bentley [1] uses GAs to evolve shapes for solid objects directed by multiple fitness measures. This structural engineering problem is known as "structural topology optimization." A similar area was pursued by Jakiela, who represented structural topology with GA chromosomes [4][6]. His approach is based on converting the chromosome into topology, evaluating its fitness and structural performance, and then assigning a fitness value to the chromosome. Later work of Jakiela represents a specification-based design evaluation method and how it is applied to optimization using GAs [5]. Eric Jones, in his Ph.D. thesis, successfully applied GAs to evolution of antennas and logic circuits. [8] He developed a special grammar, so each valid sentence in this grammar represents a valid antenna design and encoded sentences as chromosomes to perform genetic optimization on them.

One of the most successful attempts to apply GAs to the task of Lego generation was made by J. Pollack and P. Funes.[9] Their work used networks of the torque propagation to evaluate structures and genetic programming (rather then a genetic algorithm) operators to perform optimization. Also they used an assembly tree to represent Lego structures. According to Pollack it was one of the limitation factors of their work. In our research we are trying to address this particular limitation.

3 TECHNICAL APPROACH

3.1 PROBLEM FORMULATION

The main contribution of this research is not in the genetic algorithm itself, but rather in its application to the practical task of Lego design generation. We have selected Lego assembly because it represents a sufficiently complex, multi-disciplinary design domain that includes a wide variety of realistic engineering constraints and the domain is sufficiently discrete as to be tractable. This problem has great practical value: first, with growing popularity of Lego robot competitions it is very beneficial to have a tool for design generation or optimization, second, generation of Lego assemblies is closely related to the generation of the real engineering artifacts, so results of this research may scale to larger engeneering problems.

3.2 REPRESENTATION OF LEGO ASSEMBLIES

We use assembly graphs to represent Lego assemblies. Representing Lego designs as a mechanical assembly graph has a number potential of advantages over the assembly tree approach suggested in earlier research [9]. A labeled assembly graph is more expressive and can represent greater variety of Lego assemblies including kinematic mechanisms as well as static structures [13]. The nodes of the graph will represent different Lego elements and the edges of the graph will represent connections between elements. Another problem that we were facing was the absence of the notation for describing valid Lego assemblies. We developed a graph grammar, similar to the one described by Schmidt [14][15], to

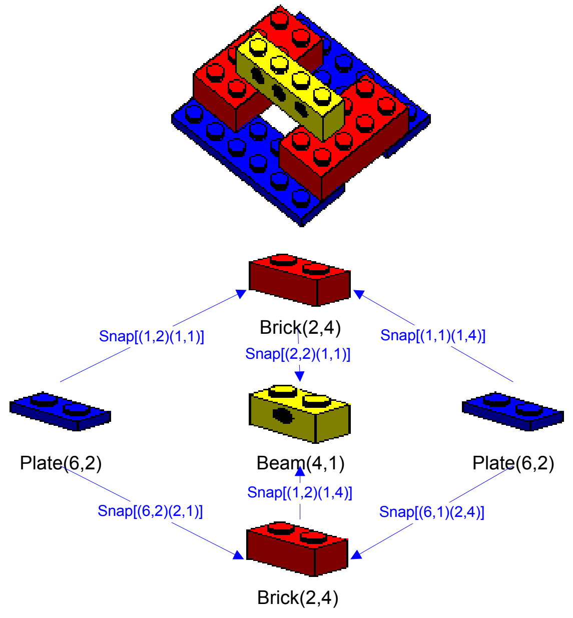


Figure 1. Example Lego structure with assembly graph.

define valid 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 non-empty set *N(G)* of nodes *n*, representing Lego elements and set *E(G)* of edges *e*, representing connections and relations between those elements [2]. The node label contains the type and the parameters of the element. For now, our program can operate only with 3 types of Lego elements, namely **Beam**, **Brick,** and **Plate**. We will combine these 3 types under the category **Block**.

Number and nature of parameters specified in the label depends on the type of element. For **Beam**, **Brick,** and **Plate** these parameters are the number of pegs on the element in *X* and *Y* dimensions. We have created a labeling scheme for the elements of type **Axle**, **Wheel** and **Gear** and are currently working on introducing it into the program.

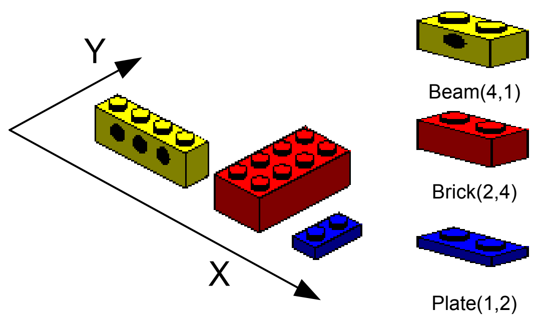


Figure 2. Examples of Lego blocks (left) with labeled nodes (right).

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* since 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 provides connection surfaces. The Edge label contains the connection type *Snap* and a Peg-Pair which is used to define the Pair of corresponding pegs in the connection [(*PozX1, PozY1); (PozX2, PozY2)]*. (*PozX1, PozY1)* defines the peg on the **Block**, which provides pegs, and (*PozX2, PozY2)* defines the peg on the **Block**, which provides connection surface. This means that the peg on the block *Snap* is pointing to always defined second in the Peg-Pair.

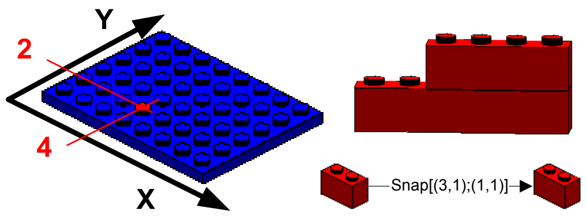


Figure 3. Example of the Peg coordinates and Snap connection.

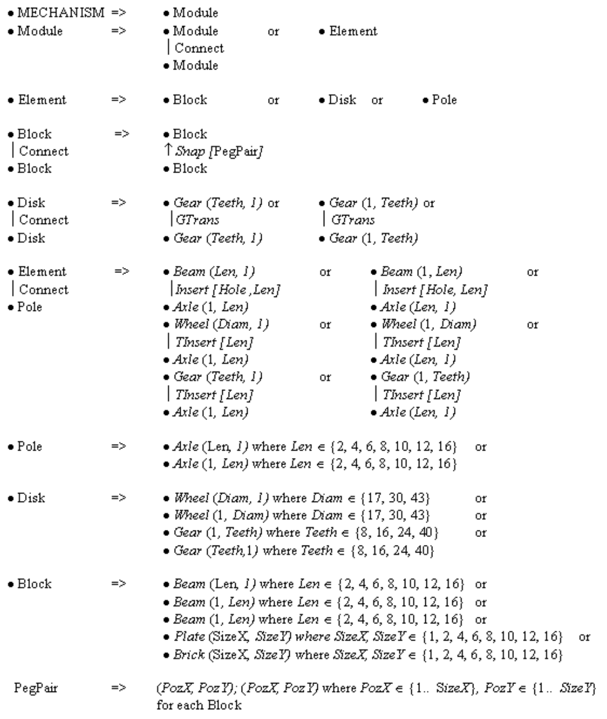
3.3 LEGO GRAMMAR.

A parameterized context-sensitive directional graph grammar was designed to handle three-dimensional structures assembled from 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*, *PozX*, *PozY*, *SizeX*, *SizeY*, *Len*, *Diam*, *Teeth*, *Hole* (,), [,], ;}. Starting word of the grammar is {*MECHANISM*}. Terminal words of the grammar are {↑*Snap*, │*Insert*, │*TInsert*, │*GTrans*, •*Beam*, •*Brick*, •*Plate*, •*Battery*, •*Motor*, •*Wheel*, •*Gear*, •*Axle*, *SizeX*, *SizeY*, *Len*, *PozX*, *PozY*, *Diam*, *Teeth*, *Hole* (,), [,], ;}. Terminal words "*()[],;*" are used only to make sentences easier to read, so most often they will be ignored on the derivation and syntax trees.

Terminals *PozX, PozY, SizeX, SizeY, Len, Diam, Hole* and *Teeth* are parameters. Meaning of the parameters *SizeX, SizeY, PozX*, and *PozY,* 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* used to specify the number of pegs on the Lego *Beam* or the length of an *Axle* measured in the peg

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



```
• MECHANISM =>   • Module
• Module    =>   • Module           or    • Element
                 │Connect
                 • Module

• Element   =>   • Block            or    • Disk   or    • Pole

• Block     =>   • Block
│Connect         ↑Snap [PegPair]
• Block          • Block

• Disk      =>   • Gear (Teeth, 1) or     • Gear (1, Teeth) or
│Connect         │GTrans                  │GTrans
• Disk           • Gear (Teeth, 1)        • Gear (1, Teeth)

• Element   =>   • Beam (Len, 1)        or    • Beam (1, Len)        or
│Connect         │Insert [Hole ,Len]          │Insert [Hole, Len]
• Pole           • Axle (1, Len)              • Axle (Len, 1)
                 • Wheel (Diam, 1)      or    • Wheel (1, Diam)      or
                 │TInsert [Len]               │TInsert [Len]
                 • Axle (1, Len)              • Axle (Len, 1)
                 • Gear (Teeth, 1)      or    • Gear (1, Teeth)
                 │TInsert [Len]               │TInsert [Len]
                 • Axle (1, Len)              • Axle (Len, 1)

• Pole      =>   • Axle (Len, 1) where Len ∈ {2, 4, 6, 8, 10, 12, 16}   or
                 • Axle (1, Len) where Len ∈ {2, 4, 6, 8, 10, 12, 16}

• Disk      =>   • Wheel (Diam, 1) where Diam ∈ {17, 30, 43}        or
                 • Wheel (1, Diam) where Diam ∈ {17, 30, 43}        or
                 • Gear (1, Teeth) where Teeth ∈ {8, 16, 24, 40}    or
                 • Gear (Teeth,1) where Teeth ∈ {8, 16, 24, 40}

• Block     =>   • Beam (Len, 1) where Len ∈ {2, 4, 6, 8, 10, 12, 16}   or
                 • Beam (1, Len) where Len ∈ {2, 4, 6, 8, 10, 12, 16}   or
                 • Beam (1, Len) where Len ∈ {2, 4, 6, 8, 10, 12, 16}   or
                 • Plate (SizeX, SizeY) where SizeX, SizeY ∈ {1, 2, 4, 6, 8, 10, 12, 16}   or
                 • Brick (SizeX, SizeY) where SizeX, SizeY ∈ {1, 2, 4, 6, 8, 10, 12, 16}

PegPair     =>   (PozX, PozY); (PozX, PozY) where PozX ∈ {1.. SizeX}, PozY ∈ {1.. SizeY}
                 for each Block
```

Figure 4. Examples of Lego grammar rules.

sizes. *Hole* defines the 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.*

$$PozX \in \{1 .. SizeX\},$$

$$PozY \in \{1 .. SizeY\}$$

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

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

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

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

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

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

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

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

Module is representing the number of Elements connected together. Element represents a single Lego piece. Grammar includes three categories of Lego elements, namely *Block*, *Disk* and *Pole*.

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

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

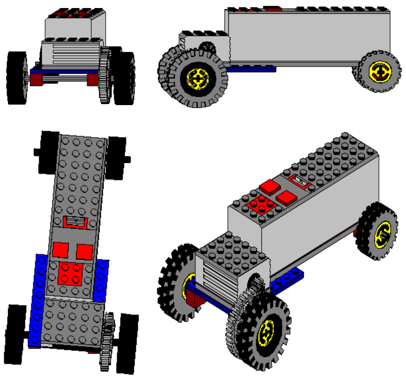


Figure 5. Example of the Lego car.

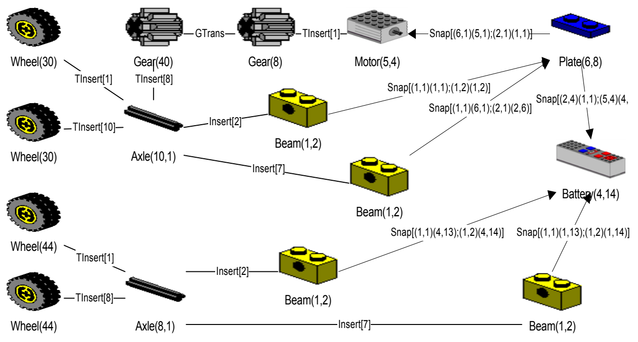


Figure 6. Assembly graph for the Lego car.

We have developed specifications on representation of wheels gears and axles and their connections an example of which is the assembly graph in Figure 6 for the Lego car in Figure 5. Having a Lego language aided us in classification of the Lego blocks and connections. For now we used the developed notation only to formally define the requirements documentation. In the future we plan to introduce another level of abstraction and represent Lego mechanisms as a sentence in a language of Lego assemblies, rather than a graph which makes it easier to validate the assembly against grammar rules [8]. A portion of our Lego graph grammar is shown in Figure 4.

3.4 GA ENCODING SCHEME

In order to perform a GA-based optimization on the population of assembly graphs, we first have to define a way to encode the assembly graph as a chromosome. Currently, the chromosome is represented by a combination of two data structures: array containing all nodes $N(G)$ and the adjacency hash table containing all edges $E(G)$ with corresponding string keys. This array is called the *genome*, and individual elements of the array called genes. Genome defines what Lego elements will compose the structure. Key value of the hash table is used to represent γ function of the graph G, and defines the

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)]

Figure 7. Chromosome of the example structure.

position and direction of an edge. Key *"1>3"* is equivalent to key *"3<1"* and means that the edge is located between nodes 1 and 3 and directed to node number 3. Hash table defines the way Lego elements will be connected together.

3.5 GENETIC OPERATIONS ON GRAPHS.

3.5.1 Initialization

The initial population is generated at random. First ten nodes of random types are generated with random dimensions. Then, 9 to 13 edges 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.

3.5.2 Mutation

In order to provide the balance between the exploration vs. exploitation mutation operator is applied with a constant low probability. Mutation operator works on the genome array of the mutated chromosome. After gene was selected for the mutation it is simply replaced with a Lego element of the same type and random size. Some edges can become invalid after the element was mutated. In this case these edges are reinitialized at random. Obviously Mutation has limited ability to change the structure on the graph and works mostly on the nodes themselves. We are planning to introduce mutation that alters edges and further the small sub-graphs.

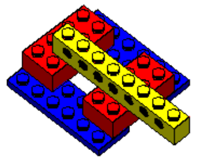


Figure 8. Sample structure with a mutated beam.

3.5.3 Crossover

As in the majority of messy genetic algorithms, 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 independent randomly selected points. Then, tail parts of the chromosomes spliced with head parts of the other chromosome. Since selection of the crossover point is independent for each chromosome, it is obvious that chromosome length can vary during the evolution process, to allow for evolution of the assemblies with different number of elements. 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 the number less than *P* are considered the head segment and all genes with the number greater or equal to *P* are considered a tail segment.

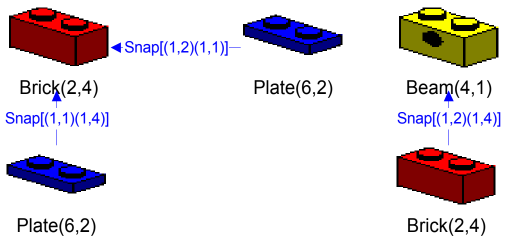


Figure 9. Sample structure after Cut operator was applied at the point 3.

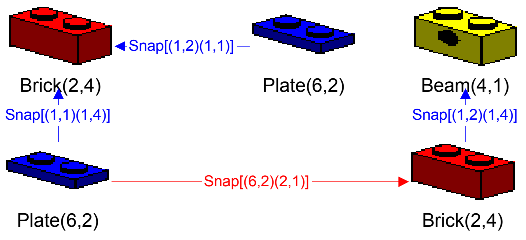


Figure 10. Head and tail sub-graphs spliced with one random edge.

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 sub-graphs are joined with a small number of randomly generated edges.

3.5.4 Evaluation Function

Each structure has a number of attributes, such as 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. Generally, evaluation functions were created according to the following form:

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

Where P_i is the weight function, which represents the importance of evaluated parameter as follows: P_i for most critical parameters, P_j, for less important and P_k for the least important.

$$P_i = x_i,\ P_j = x_j^{1/2},\ P_k = x_k^{1/4}$$

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

3.5.5 Handling Over-specified/Under-specified Chromosomes

As in most of the messy genetic algorithms there is a chance of over-specified or under-specified chromosome, generated during the evolution process [3]. Under-specification 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 sub-graph containing the *0*-th node are selected to be dominant. The submissive sub-graph is not deleted from the chromosome, but is ignored in most calculations. Figure 9 can demonstrate an example of the under-specified chromosome (i.e. a disjoint Lego assembly graph). On the other hand an over-specified 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 are connected by the set of edges, which imply two different locations for the same node. In the first case, the node which was assigned the location first is marked as dominant, and the node which was assigned location second is marked as submissive and ignored in most calculations. In the case when different edges imply different locations for the same block edge which traversed first is given priority and other edges are marked as submissive and ignored.

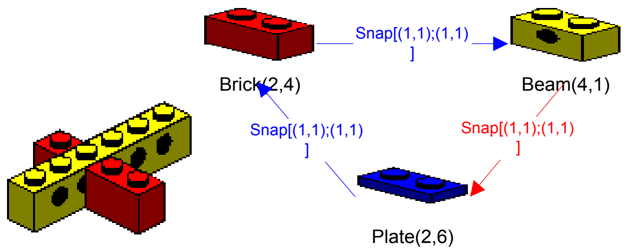


Figure 11. Example of the blocks sharing same physical space (left) and infeasible connection edges (right).

4 EXAMPLES AND CURRENT RESULTS

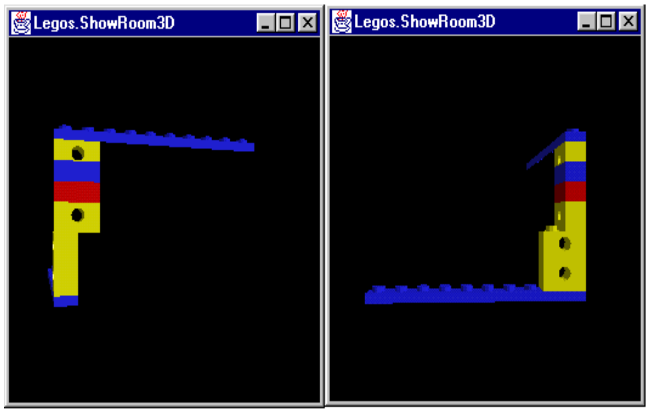
4.1 SYSTEM DESCRIPTION

Our system was extended from *sGA* system originally created by Hartley [16] and written in the Java programming language. Java3D and VRML97 were used in order to create a visualizer to monitor Lego structures as they evolve. The system supports one-point crossover. We are planning to introduce uniform and N-point crossover in the future. The user can choose from proportional, rank, universal stochastic sampling, sexual, sigma scaling, and Boltzman selection techniques. Other parameters to the system include mutation and crossover rates, and population size. Although the current system can only handle static structures composed of block type elements, the general approach can be applied to much more elaborate kinematic mechanisms.

4.2 EXPERIMENTS

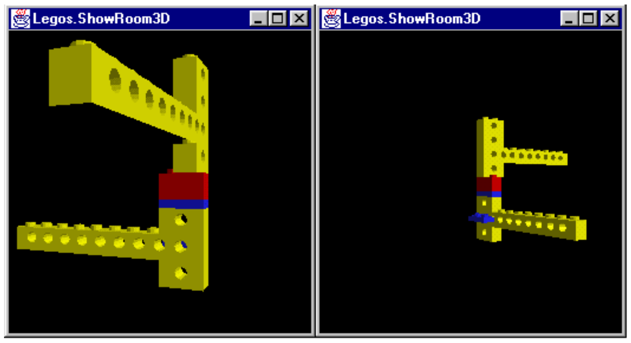
4.2.1 Evolving 10 by 10 by 10 Structure

Figure 12 demonstrates the result of 1000 generations of evolution of a 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 the generation 895 and has the sizes 10 by 10 by 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 13. This structure as discovered by the GA in the generation 3367 and it is a little bit heavier but it has perfect 10 by 10 by 10 size.



```
number of crossovers, mutations and reinitializations: 34695, 2459 and 0
Simulation completed in 1000 generations and 69.16 seconds
Best_member (generation= 895 ):
  Nodes: Brick 1x2 Beam 2x1 Plate 1x2 Beam 1x2 Beam 1x2 Plate 1x2 Plate 10x1
Pl
ate 2x1 Beam 2x1 Plate 1x10
  Nodes: 10 Size: 10.0x10.0x6.8 Weight: 52.79999923706055
   fitness= 0.09554419512063303
Close ShowRoom window to exit.
```

Figure 12. . Example of the graphical and text output of our system.



```
Simulation completed in 10000 generations and 1030.24 seconds
Best_member (generation= 3367 ):
  Nodes: Beam 1.0x10.0 Beam 1.0x2.0 Plate 2.0x1.0 Beam 1.0x2.0
Brick 2.0x1.0 Beam 2.0x1.0 Beam 10.0x1.0 Beam 2.0x1.0 Plate
2.0x1.0 Beam 1.0x2.0
  Nodes: 10 Size: 10.0x10.0x10.000000149011612 Weight:
87.99999904632568
   fitness= 0.24613479430155352
Close ShowRoom window to exit.
```

Figure 13. Example of the graphical and text output of our system.

4.2.2 Evolving Pillar-Like Structure

Another line of experiments is shown in Figure 14, where we were evolving a pillar-like structure. The goal was to make an assembly with 4 by 2 base in x-y dimension and 20, 40 and 60 length in 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

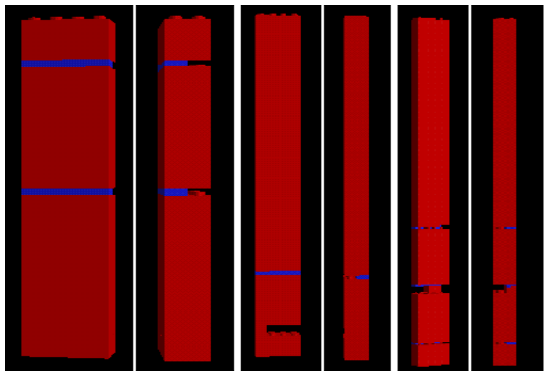


Figure 14. Example of the graphical and text output of our system.

intervals. On average solution was discovered within 5000 generations. All the structures have desired size, and have very few defects.

4.3 LIMITATION AND FUTURE WORK.

Currently, our research is in its early stages. We are planing to introduce more types of Lego elements and connections. We have specification for encoding elements of the type Wheel, Gear and Axle as well as connections between them, but we still have to implement it in code. Other types of Lego elements such as Motors, Batteries and their connections have to be described by the grammar and implemented later. We are also planning to develope more realistic physical models, to help us better evaluate the structures. All of these enhancements will help us generate Lego structures more fitted to perform a specific task.

There are also a number of improvements to the genetic operators themselves. For example, creating a mutation operator, which would alter small sub-graphs, will help algorithm to explore areas of the fitness landscape neighboring to the solution.

For now the cut operator was applied at random points often separating elements which should work together into two different sub-graphs. We are planing to introduce a notion of clusters: highly interconnected sub-graphs loosely coupled together, which would correspond to the separate modules or "organs." System must promote crossover on the cluster boundaries, and demote crossover, which brakes up a cluster.

Another improvement, which can significantly speed up the search, is in using guided or seeded initialization. This would mean introducing absolute as well as probabilistic rules and applying them during the generation of the initial population, or injecting pre-build mechanisms in to the initial population.

5 CONCLUSIONS

This paper has 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 and evolution by Genetic Algorithms. This graph-based approach is unlike other systems for Lego design evolution [10], and we believe that this assembly graph-based representation scheme is one of the most general ways of representing the assembly, and in this way provides a more flexible means to represent a wide variety of mechanisms for use with GAs. Another unique feature of our research is the development of a graph grammar for use in representing Lego assemblies. Although we used it in all of our requirements documentation for defining connectivity of the Lego elements, all of the rules were hard-coded into the system during the implementation cycle. We believe that Lego sentences, rather than graphs on the code level, can bring this approach to a new level.

Acknowledgments

This work was supported in part by National Science Foundation (NSF), nowledge 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 National Institute of Standards and Technology (NIST) Grant 60NANB7D0092.

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

References

1. P. Bentley (ed.) (1999). *Evolutionary Design by Computers.* Morgan Kaufmann Publishers; ISBN: 155860605X.
2. S. Chase (1996). *Representing designs with logic formulations of spatial relations.* Workshop Notes, Visual Reasoning and Interaction in Design.
3. D. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub Co; ISBN: 0201157675.
4. M. Jakiela (1998). *Structural Topology Design with Genetic Algorithms*, Computer Methods in Applied Mechanics and Engineering.
5. M. Jakiela, D. Wallace, W. C. Flowers (1996). *Design search under probabilistic specifications using genetic algorithms*, Computer-Aided Design, V 28, N 5, pp. 405-421.
6. M. Jakiela, J Duda (1997). *Gereation and classification of structural topologies with genetic algorithm speciation*, Journal of Mechanical Design, V 119, Number 1, pp. 127–130.
7. M. Jakiela, C. Chapman, K. Saitou (1994). *Genetic algorithms as an approach to configuration and topoly design*, Journal of Mechanical Design, V 116.
8. E. Jones (1999). *Genetic design of antennas and electronic circuits*, Ph.D. Thesis, Department of Electrical and Computer Engineering Duke University.
9. J. Pollack, P. Funes (1997)., *Computer Evolution of Buildable Objects.* Fourth European Conference on Artificial Life, P. Husbands and I. Harvey, eds., MIT Press pp 358-367.
10. J. Pollack, P. Funes (1998). *Evolutionary Body Building: Adaptive physical designs for robots.* Artificial Life 4, pp. 337-357.

11. J. Pollack, *P.* Funes (1998). *Componential Structural Simulator.* Brandeis University Department of Computer Science Technical Report CS-98-198.

12. J. Pollack, R. Watson, S. Ficici (1999). *Embodied Evolution: Embodying an Evolutionary Algorithm in a Population of Robots.* 1999 Congress on Evolutionary Computation. Angeline, Michalewicz, Schoenauer, Yao, Zalzala, eds. IEEE Press, 335-342.

13. L. Schmidt, H. Shi, S. Kerkar (1999). *The "Generation gap": A CSP Approach linking function to form grammar generation*, Proceedings of DETC99, ASME Design Engineering Technical Conference.

14. L. Schmidt, H. Shetty, S. Chase (1996). *A Graph Grammar Approach for structure synthesis of Mechanisms*, Journal of Mechanical Design.

15. L. Schmidt, J. Cagan (1998). *Optimal Configuration Design: An Integrated approach using grammar*, Transactions of the ASME, V 120, N 2, March.

16. URL:http://www.mcs.drexel.edu/~shartley/ConcProgJava/GA/Simple.