Evolutionary Learning of Novel Grammars for Design Improvement

John S. Gero, Sushil J. Louis [1] and Sourav Kundu

Key Centre of Design Computing
Department of Architectural and Design Science
University of Sydney
NSW 2006 Australia.
john@archsci.arch.su.edu.au

Abstract

This paper focusses on that form of learning which relates to exploration, rather than generalization. It uses the notion of exploration as the modification of state spaces within which search and decision making occur. It demonstrates that the genetic algorithm formalism provides a computational construct to carry out this learning. The process is exemplified using a shape grammar for a beam section. A new shape grammar is learned which produces a new state space for the problem. This new state space has improved characteristics.

1 Introduction

Design can be considered a purposeful, constrained, decision making, exploration and learning activity. Decision making implies a set of variables, the values of which have to be decided. Search is the common process used in decision making. Exploration here is akin to changing the problem spaces within which decision making occurs. Learning implies a restructuring of knowledge as opposed to restructuring of facts. Searching in design is concerned with restructuring of facts while exploration is a learning process which restructures the knowledge used in searching. The designer operates within a context which partially depends on the designers perception of purposes, constraints and related contexts. These perceptions change as the designer explores the emerging relationships between putative designs and the context as the designer learns more about possible designs (Gero 1992).

One useful framework for design uses the concept of design prototypes (Gero 1987,1990). This divides a design state space into three subspaces corresponding to a space of structures, a space of behaviors associated with structures and functions and a space of functions associated with behaviors. Figure 1 shows the mappings between these subspaces. Generating new designs can be represented as a process which changes one of the subspaces (usually the structure space is changed). There are two interesting classes of change to state spaces: addition and substitution. Addition, which can be achieved by adding variables, results in a new state space, S_n, that subsumes the original space, S_0, that is: $S_0 \subset S_n$

[1] On leave from the Department of Computer Science, Indiana University.

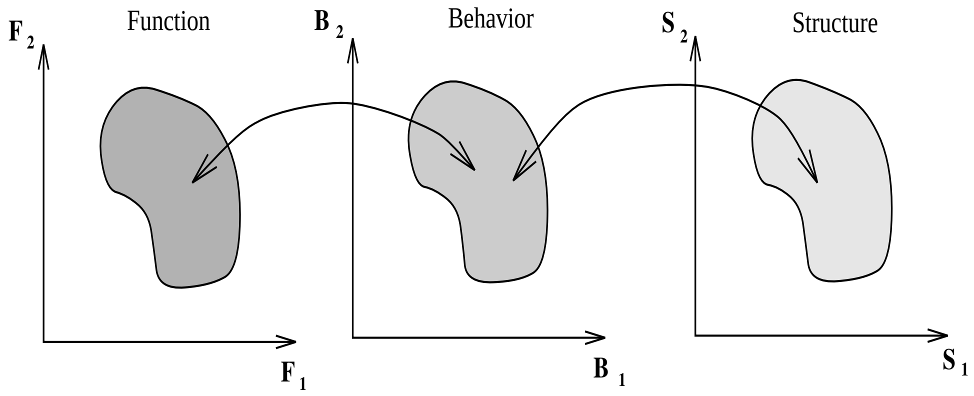

Figure 1: Structure, Behavior and Function spaces and the mappings between them.

and $S_0 \bigcap S_n \neq \phi$ (Figure 2). Substitution, which involves the replacement of some variables by other variables, results in a space, S_n, that is different from the original, S_0, such that $S_0 \not\subset S_n$ (Figure 3). The notions of additive

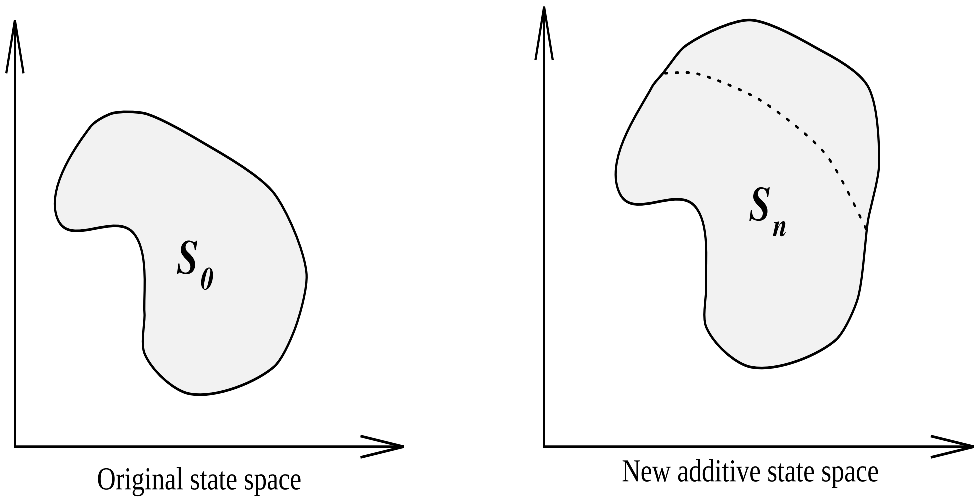

Figure 2: The effect of additive processes on a state space.

processes and substitutive processes also apply to design schemas resulting in additive schemas that subsume the original one and substitutive schemas that only include parts of the original and contain new elements (Gero 1992). These are depicted in Figures 4 and 5.

In this paper we concentrate on the learning aspect of design, focussing on that form of learning which relates to exploration, that is, modifying the problem spaces within which decision making occurs.

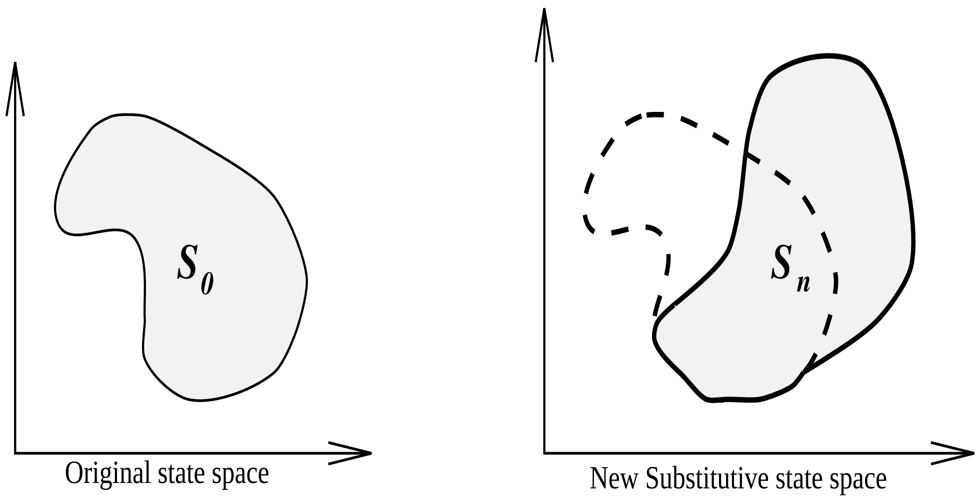

Figure 3: The effect of substitutive processes on a state space.

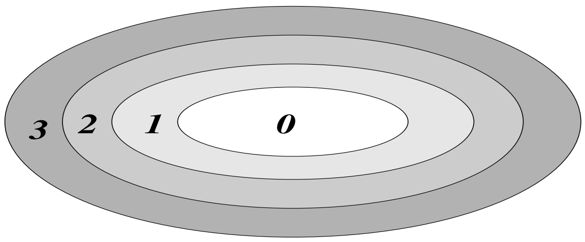

Figure 4: The effect of additive processes on design schemas, changing from original schema, 0, to schemas 1, 2 and 3.

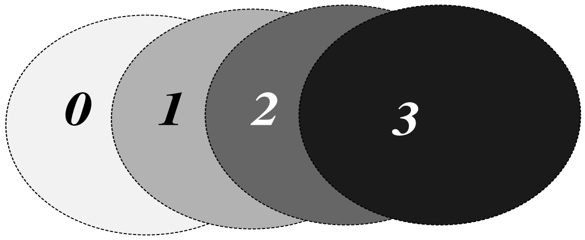

Figure 5: The effect of substitutive processes on design schemas, changing from original schema, 0, to schemas 1, 2 and 3.

2 Machine Learning

2.1 A Different View

Machine learning is comprised of principally four paradigms (Carbonell 1990, Mitchell et al 1986): the inductive paradigm, the analytic paradigm, the genetic paradigm, and the connectionist paradigm. These can be broadly categorized as "learning to perform old tasks better using available tools". Thus, the main emphasis is on improving previous task schedules and routines to achieve certain given goals. Inductive learning is a good example of this. The system is provided with a set of instances and it produces generalizations of those instances grouped under broad concepts that explain the instances. Two main types of inputs are: examples and observations. Examples are instances that are classified by the teacher as positive or negative instances. Observations are instances that are not classified by the teacher. Thus if some conceptual examples are given to the learning program, it learns rules that could produce such examples. Quinlan's ID3 (Quinlan 1979,1986) and Reich's Bridger (Reich 1991) are examples of this category of machine learning.

We propose a second view of learning which can be summarised as:

> learning to restructure knowledge to produce novel results that could not be achieved using the current knowledge.

Learning is always concerned with improving the quality of knowledge, traditionally from examples. A number of different teleologies drive machine learning not all of which are applicable in design. The foci in machine learning in design include concept formation (Maher and Li 1992, 1993) learning high level relationships (Rao et al 1991, Gunaratnam and Gero 1993), and increasing the efficacy of available knowledge (Arciszewski et al 1987). The teleology of the learning approach adopted here is to improve the quality of the knowledge to produce improved designs. It is concerend with restructuring of the design knowledge which is already in the form of generalizations.

Grammar induction (Mackenzie 1989,1991) is an approach which can be characterized in these terms. An important difference between the grammar induction technique and the one we are proposing here, is that, the state space of structures produced by the induced grammar always has the example set as its proper subset. This matches the the notion of "additive state spaces" which can also be achieved by the introduction of new variables in the design (Gero and Kumar 1993). The view of learning that we present here can be mapped onto the notion of "substitutive state spaces". The implication of this substitutive view is that some existing variables are deleted and some new ones are added. There is no nexus between the number of existing variables deleted and the number of new ones added. In the existing notion of learning in design the purpose of the learning activity is to predict a class into which a given example can be classified so as to produce designs conceptually the same as the examples used to learn (McLaughlin and Gero 1987, Mackenzie and Gero 1987). Thus the state spaces are very much fixed by the jurisdiction of the given examples.

The class of learning we are presenting here restructures the design knowledge so as to produce new points in design spaces which lie outside the realm of the state spaces defined by the given examples. These points are discovered during an evolutionary process by the creation and exploration of possibly novel state spaces using the restructured knowledge. The learning program is *not* presented with examples, rather it is presented with a set of rules in a grammar which, when executed, produces examples. The state spaces are not defined by the given examples but by the grammar that produces those examples. As a new grammar is learned the state spaces change, perhaps drastically. This type of system can be seen as more of a generative or formation system rather than a classification system. Thus the system tries to achieve better designs by learning to restructure knowledge which is capable of producing novel designs which *could not* be produced with the original knowledge. We give a short introduction to shape grammars and genetic algorithms which we use in our learning process.

2.2 Shape Grammars

Shape grammars were introduced into the architectural literature as a formal method of shape generation. They provide a recursive method for generating shapes and are similar to phrase structure grammars, but defined over alphabets of shapes and generate languages of shapes (Stiny and Gips 1978). A set of grammatical rules map one shape into a different shape. These rules define the set of possible mappings or transformations. More formally, a shape grammar is the quadruple (V_t, V_m, R, I). Where V_t is a set of terminal shapes or terminals and V_m a set of nonterminal shapes or markers. V_t and V_m provide the primitive shape elements of a shape grammar. R is a set of rules consisting of two sides, each side of which contains members of $V_t \bigcup V_m$. If the left hand side of a rule matches a shape, applying the rule results in replacing the matching shape with the right hand side of the rule. I is the initial shape, a subset of $V_t \bigcup V_m$ and starts the shape generation process. This models a design system where the rules embody generalized design knowledge and a sequence of rule applications generates a design.

2.3 Genetic Algorithms

Genetic algorithms (GAs), originally developed by Holland (1975), model natural selection and the process of evolution. Conceptually, GAs use the mechanisms of natural selection in evolving individuals that, over time, adapt to an environment. They can also be considered a search process, searching for better individuals in the space of all possible individuals. In practice, individuals represent points in a state space, while the environment provides a measure of "fitness" that helps identify better individuals. Genetic algorithms are a robust, parallel search process requiring little information to search effectively. As such they are well suited to the task of exploring and learning about large and complex design spaces.

2.4 Description of the Problem

We model routine design with a fixed set of shape grammar rules and encode the possible execution order (application sequence) of these rules for manipulation by the genetic algorithm. The set of optimal structures for this fixed grammar defines a space of feasible solutions corresponding to a space of behaviors. The goal is to find the execution order of the grammar rules which will optimize a set of behaviors. In this fixed scheme, additive and substitutive processes are absent. However, when in addition to the application sequence, we allow the grammar itself to be encoded for manipulation by the genetic algorithm, we learn new grammars and associated rule application sequences to improve on the best possible designs that could be generated by the fixed grammar. That is, instead of optimizing a plan of application of some fixed set of rules, the computational model learns new rules and optimizes application sequences for those new rules to generate novel and 'more optimal' solutions. By more optimal we mean that the optimal behaviors produced by the application of the new rules are better than those produced by the application of the original rules. To understand how the genetic algorithm learns new grammars we provide a brief introduction to genetic algorithms.

3 Genetic Algorithms

3.1 Introduction to Genetic Algorithms

The motivational idea behind GAs is natural selection implemented through selection and recombination operators. A population of "organisms" (usually represented as bit strings) is modified by the probabilistic application of the genetic operators from one generation to the next. The basic algorithm where $P(t)$ is the population of strings at generation t, is given below.

$t = 0$
initialize $P(t)$
evaluate $P(t)$
while (termination condition not satisfied) **do**
begin
    select $P(t+1)$ **from** $P(t)$
    recombine $P(t+1)$
    evaluate $P(t+1)$
    $t = t + 1$
end

Evaluation of each string which corresponds to a point in a state space is based on a fitness function that is problem dependent. This corresponds to the environmental determination of survivability in natural selection. Selection is done on the basis of relative fitness and it probabilistically culls from the population those points which have relatively low fitness. Recombination, which consists of mutation and crossover, imitates sexual reproduction. Mutation, as in natural systems, is a very low probability operator and just flips a specific bit. Crossover in contrast is applied with high probability. It is a structured yet stochastic operator that allows information exchange between points. Simple crossover is implemented by choosing a random point in the selected pair of strings and exchanging the substrings defined by that point. Figure 6 shows how crossover mixes information from two parent strings, producing offspring made up of parts from both parents. We note that this operator which does no table lookups or backtracking, is very efficient because of its simplicity.

3.2 Genetic Algorithm Encodings

Understanding how to represent a problem and any domain knowledge of the problem in a genetic algorithm requires 1) knowing which properties of a search space GAs use to guide their exploration and 2) the need to produce viable offspring during crossover. A smidgeon of GA theory clears the way.

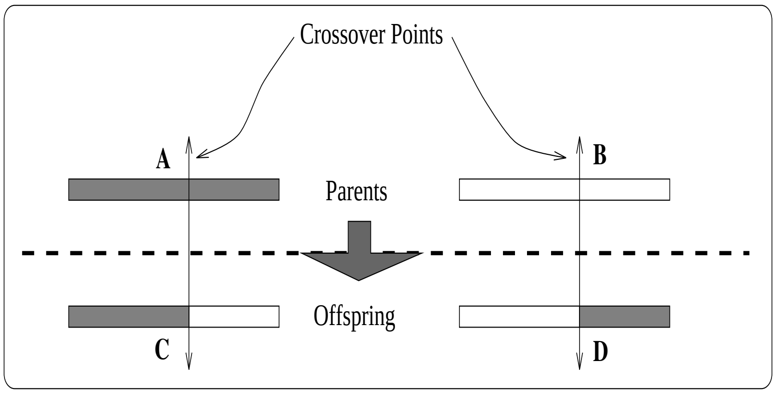

Figure 6: Crossover of the two parents A and B produces the two children C and D. Each child consists of parts from both parents which leads to information exchange.

Crossover causes genotypes (an encoded individual) to be cut and spliced. This means that instead of considering the fate of individual strings in analyzing a genetic algorithm, we must consider the substrings created and manipulated by crossover. These substrings define regions of the search space and are called schemas[2]. More formally, a schema is a template that identifies a subset of strings with similarities at certain string positions. Holland's *schema theorem* is fundamental to the theory of genetic algorithms. For example consider binary strings of length 6. The schema 1**0*1 describes the set of all strings of length 6 with 1*s* at positions 1 and 6 and a 0 at position 4. The "*" denotes a "don't care" symbol which means that positions 2, 3 and 5 can be either a 1 or a 0. Although we only consider a binary alphabet this notation can be easily extended to non-binary alphabets. The *order* of a schema is defined as the number of fixed positions in the template, while the *defining length* is the distance between the first and last specific positions. The order of 1**0*1 is 3 and its defining length is 5. The *fitness* of a schema is the average fitness of all strings matching the schema.

[2]Different from design schemas

The genetic algorithm therefore implicitly processes schemas. Because of the bias introduced by selection and crossover, certain types of schemas rapidly increase their proportions in a population. Not only does the rate of increase depend on their fitness, it also depends on the syntactic properties of order and defining length. This is of great importance when incorporating domain knowledge into the bit strings that a GA manipulates. The schema theorem proves that relatively short, low-order, above average schemas get an exponentially increasing number of copies in subsequent generations. This leads to the building block hypothesis which states that genetic algorithms work near-optimally by combining short, low-order, high fitness schemas called *building blocks* (Goldberg 1989). Thus when encoding domain knowledge, we should choose an encoding that reflects a GA's bias toward short, low-order building blocks. This bias is critically affected by the crossover operator. The plethora of crossover operators, each with its own bias, implies analyzing and carefully matching an operator's bias in the encoded representation of domain knowledge. In practice, many encodings work well because of the large number of schemas and a GA's reliance on fitness information.

Non-viable offspring are produced when crossover results in individuals that do not belong to the search space of interest. In the context of grammars, if an individual represents a sequence of rule applications resulting in a shape, it may call for the application of a rule that does not exist in the grammar. An example clarifies the problem.

Consider the grammar in Figure 7. There are four classes of composition rules, one associated with each of the cells labeled A, B, C, D. In total there are eight rules in these four classes. If we encode the rules of class 1 as the numbers from $0-2$, the rules of class 2 as the numbers from $3-5$, the rule of class 3 as 6 and the rule of class 4 as 7, then it is easy to produce a binary string for the genetic algorithm to work on. Fixing the number of rule applications results in a fixed length binary string. As explained in section 3.1, genetic algorithms normally use binary representations/encodings of fixed length. Consider an example in which we fix the number of rule applications at 3, then the string

$$2, 3, 6$$

corresponds to successively applying rules $2, 3$ and 6. Before we convert this into a binary string, the question of how many binary digits, or bits, are required for each decimal number needs to be answered. Each position in the string can have values ranging from 0 to 7 corresponding to each of the rules in the grammar (Figure 7). Therefore we use three (3) bits to represent the 8 possible rules. The string $2, 3, 6$ would then be represented as the binary string

$$010, 011, 110$$

for the genetic algorithm. The least significant bit is on the right and the commas are for clarity only and would not appear in the actual string that the genetic algorithm manipulates. The binary string

$$001, 010, 011$$

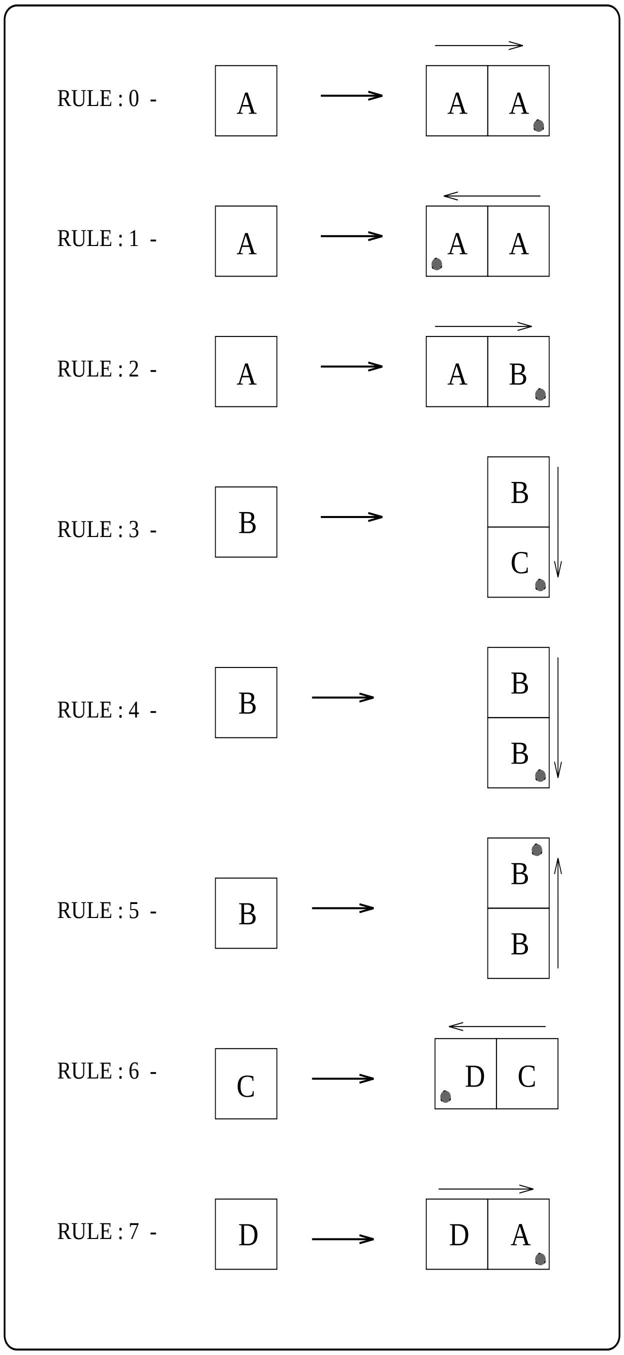

Figure 7: These eight rules comprise the fixed shape grammar.

represents another sequence of rule applications. With these strings as parents, crossover can now produce non-viable offspring as shown below

$$\begin{array}{lll} \text{Parents} & 0100111 \mid 10 & 0010100 \mid 11 \\ & \Downarrow & \\ \text{Offspring} & 0100111 \mid 11 & 0010100 \mid 10 \end{array}$$

Here the first offspring $010, 011, 111$ is not viable because rule $111 = 7$ cannot be applied after rule $011 = 3$ since the left hand side of rule 7 ($\boxed{\mathrm{D}}$) does not match the right hand side of rule 2 ($\boxed{\mathrm{C}}$). There are a few approaches to tackling problems of this kind:

- Assign a penalty for such transgressions thereby lowering the fitness of the individual. This brings up the question of which penalty function to use.
- Kill the individual, not allowing it to continue. However, we lose any potentially good building blocks encoded by the individual.
- Modify the crossover operator to only produce viable offspring, potentially changing the search bias in unpredictable ways.
- Modify the encoding so that non-viable offspring cannot occur. This is the best and most elegant solution but may be difficult to implement.
- Change the algorithm.

In practice a host of factors affect which method is chosen including the nature of the problem, convenience and time available to create an encoding. We describe the learning of new grammars and the encoding used in the next section.

4 Learning Novel Grammars

4.1 Generating New Grammars

When modeling a design process using a (shape) grammar and its associated language, we are restricted by the choice of grammar. The design task in this situation is a planning task: to plan a sequence of rule applications that will generate a desired behavior from the resultant shape. In this case the structure and behavior spaces are fixed and defined by the grammar. We can encode the task for the GA by numbering the rules and representing an individual as a finite length string of these numbers as shown in the previous section. To attack the problem of learning to produce better designs, we allow the grammar itself to evolve while generating a sequence of rule applications. This is the crux of the learning process - the restructuring of the knowledge represented by the grammar. Since every grammar defines a state space, the GA now explores a number of structure spaces with a single behavior space in parallel. This expands the number of possible designs and in the case of our system, produces better, more optimal designs and associated behaviors that were not possible before, through the application of this learned grammar. Most learning techniques are concerned with producing a structure which best fits, according to one or more criteria, the examples presented. The learning process here uses this same idea but the the examples are not presented to the learning process directly. The examples appear as a result of the structure of the knowledge encoded in the grammar. The behaviors of the examples through the evolutionary process pressure the structure to change itself so as to produce examples with improved behaviors. As a result of this evolutionary process new grammars are learned.

New grammars are generated from the original grammar, through mutation and crossover of rules. That is, rules from the original grammar serve as a basis for the generation of new grammar rules and are produced by cutting and splicing the original rules. Consider the two rules labeled "parents" in Figure 8, choosing the crossover point as indicated produces the new rules labeled "offspring." Different crossover points produce different "offspring" rules leading to a number of different grammars. Recombination therefore plays an important part in the process of learning, helping to generate different grammars and thus different structure spaces to explore.

4.2 An Example - Designing a Beam Section

Using a shape grammar, our goal is to learn new grammars capable of producing new topologies of a beam section which give us optimal values of two specific properties of the section. To generate different beam sections to choose from, we have a set of square building blocks, called cells to distinguish them from building block in GAs, which build up a section when juxtaposed with one another. There is also a set of rules that govern this juxtaposition. Each cell has a label which is referred to by the composition rules of our grammar and which has a weight associated with it. Thus a certain configuration of the cells will have a beam section topology which will have two criteria to be optimized. The two properties have been chosen in such a way that a tradeoff needs to be made. A change for the better in one criterion usually leads to a change for the worse in the other.

The two properties are :

- The moment of inertia (second moment of area) of the beam section which has to be maximized (in order to improve its stiffness).
- The perimeter of the beam section which has to be minimized (in order to minimize the surface area).

We are trying to learn new grammars that, when executed, produce different configurations of the cells which have improved fitness values for both the above criteria. The moment of inertia of the beam section, when maximized, will tend to configure the cells in a way that will produce poor values for minimizing the perimeter of the beam section, and vice-versa. We arbitrarily commence with the eight rules shown in Figure 7. The number of rules is not significant in demonstrating the ideas in the learning process.

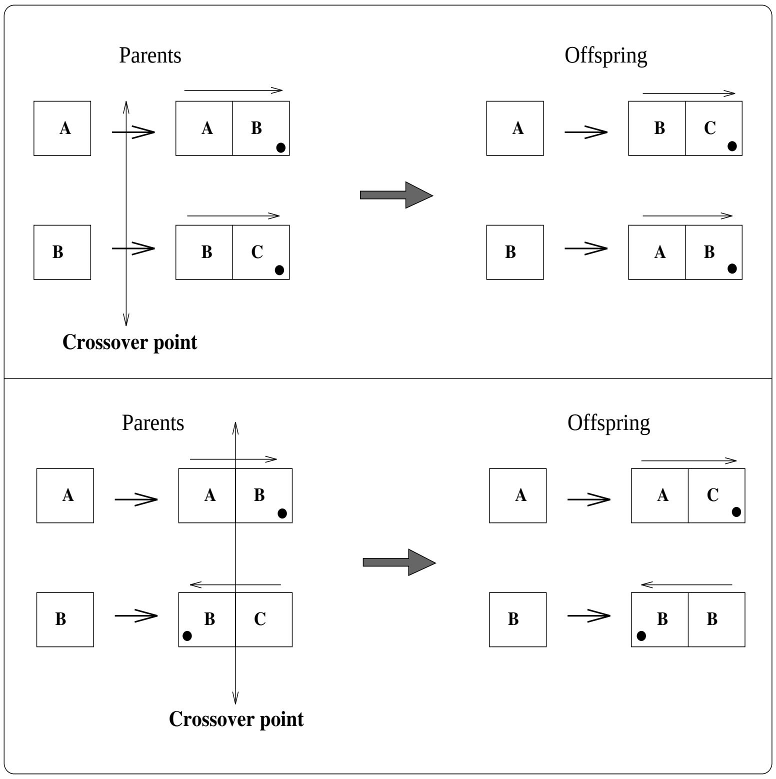

Figure 8: Generating new grammar rules by crossover, different crossover points produce different offspring rules.

Thus starting with an arbitrary label and selectively applying the rules, we can arrive at a topology for a beam section after a predetermined number of rule applications. From this grammar, shapes of interest can be derived after nine rule applications, we therefore fix the number of rule applications at nine. Similarly, the number of rule applications (or cells) is not significant in demontrating the ideas.

Weights, associated with labels, figure in the calculation of the moment of inertia simulating larger cross sectional areas, or different materials and are given below:

- $A \Rightarrow 1\ unit$
- $B \Rightarrow 2\ units$
- $C \Rightarrow 3\ units$
- $D \Rightarrow 4\ units$

These weights eventually give us a different pair of values for the two behaviors of the beam section topology that we derive with the rule applications. Thus every structure has such a pair of behavior values, and is mapped onto a two dimensional vector space with these two behaviors as the two co-ordinates. We can therefore define a set of Pareto optimal solutions from each set of new structures that we generate.

We start with the fixed grammar defined in Section 3.2. The genetic algorithm only encodes the rule application sequence in each individual. We describe the encoding chosen to solve the problem of non-viable offspring below.

We could let the numbers from $0 - 7$ represent the rules of the grammar in Figure 7. Such a representation of the problem however, results in non-viable offspring as shown in Section 3.2. To avoid the problem of non-viable offspring we change the representation/encoding of the problem for the genetic algorithm.

In the new encoding, we let the interpretation of a number *depend* on the right hand side of the last applied rule. (In the following discussion we use the numbers **0 - 7** in boldface to identify the 8 rules of our shape grammar in Figure 7.) In our encoding, a string of nine numbers specifies a shape (recall that nine rule applications are sufficient to generate interesting shapes). The first number in the string specifies the starting label, that is, whether the starting cell is of type $\boxed{A}$, $\boxed{B}$, $\boxed{C}$ or $\boxed{D}$. The eight remaining numbers specify an application sequence. Consider the string

$$0, 2, 1, 3, 0, 2, 1, 3, 1$$

The first number 0 fixes the starting cell as a cell of type $\boxed{A}$, The second number 2, now refers to the *second* rule with an $\boxed{A}$ on the left hand side. Therefore, in this encoding the number 2 in the second position of the string, refers to rule **1**, that is,

$$\boxed{A} \longrightarrow \overleftarrow{\boxed{\bullet\ A}}\boxed{A}$$

Once again only the rules with an $\boxed{A}$ on the left hand side can be applied and the next number, 1, refers to the first rule with an $\boxed{A}$ on he left hand side — rule **0**. The number 3 at the fourth position in the string, refers to the third rule with an $\boxed{A}$ on the left hand side. This is rule **2**:

$$\boxed{A} \longrightarrow \boxed{A}\overrightarrow{\boxed{B\bullet}}$$

which results in the first appearance of a $\boxed{B}$. The next number 0 now refers to only those rules with a $\boxed{B}$ on the left hand side. In this case, 0 refers to the first rule with a $\boxed{B}$ on the left hand side which is rule **3**,

$$\boxed{B} \longrightarrow \begin{array}{l}\boxed{B}\\ \boxed{C\bullet}\end{array} \downarrow$$

Since there is only one rule with a $\boxed{C}$ on the left hand side, all numbers from $0 - 3$ refer to this one rule. Therefore 2 refers to rule **6**

$$\boxed{C} \longrightarrow \overleftarrow{\boxed{\bullet\ D}}\boxed{C}$$

Similarly since there is only one rule with a $\boxed{D}$ on the left hand side all numbers from $0 - 3$ refer to this rule and 1 refers to rule **7**.

$$\boxed{D} \longrightarrow \boxed{D}\overrightarrow{\boxed{A\bullet}}$$

We end up with an A on the left hand side, so 3 refers to rule **2** again, and the last number 1 refers to rule **4** and we are done.

In summary:

- if there is an $\boxed{\mathrm{A}}$ on the left hand side
  - 0 and 3 ($00, 11$ in binary) refer to rule **2**

$$\boxed{\mathrm{A}} \longrightarrow \overrightarrow{\boxed{\mathrm{A}}\boxed{\mathrm{B}\bullet}}$$

  - 1 (01 in binary) refers to rule **0**

$$\boxed{\mathrm{A}} \longrightarrow \overrightarrow{\boxed{\mathrm{A}}\boxed{\mathrm{A}\bullet}}$$

  - 2 (10 in binary) refers to rule **1**

$$\boxed{\mathrm{A}} \longrightarrow \overleftarrow{\boxed{\bullet\,\mathrm{A}}\boxed{\mathrm{A}}}$$

- if there is a $\boxed{\mathrm{B}}$ on the left hand side
  - 0 and 3 ($00, 11$ in binary) refer to rule **3**

$$\boxed{\mathrm{B}} \longrightarrow \begin{array}{l}\boxed{\mathrm{B}}\\ \boxed{\mathrm{C}\bullet}\end{array} \downarrow$$

  - 1 (01 in binary) refers to rule **4**

$$\boxed{\mathrm{B}} \longrightarrow \begin{array}{l}\boxed{\mathrm{B}}\\ \boxed{\mathrm{B}\bullet}\end{array} \downarrow$$

  - 2 (10 in binary) refers to rule **5**

$$\boxed{\mathrm{B}} \longrightarrow \begin{array}{l}\boxed{\mathrm{B}\bullet}\\ \boxed{\mathrm{B}}\end{array} \uparrow$$

- if there is a $\boxed{\mathrm{C}}$ on the left hand side
  - $0 - 3$ (00 to 11 in binary) all refer to rule **6**

$$\boxed{\mathrm{C}} \longrightarrow \overleftarrow{\boxed{\bullet\,\mathrm{D}}\boxed{\mathrm{C}}}$$

- if there is a $\boxed{\mathrm{D}}$ on the left hand side
  - $0 - 3$ (00 to 11 in binary) all refer to rule **7**

$$\boxed{\mathrm{D}} \longrightarrow \overrightarrow{\boxed{\mathrm{D}}\boxed{\mathrm{A}\bullet}}$$

With this encoding not only do we solve the problem of non-viable offspring we also reduce the number of bits needed to encode a shape as we use only two bits per rule instead of three.

When we allow the grammar itself to evolve, we need to encode the rules within the genotype. Once again we choose an encoding that does not allow non-viable offspring. Making the grammar implicit, we encode a sequence of directions (up, down, left or right) and labels (A, B, C, or D) to specify an individual. This added flexibility means we require 6 bits to specify the next move in generating a shape, 2 bits for direction and 2 bits each for a label. The rules that we use are now of the form

$$\boxed{\mathrm{X}} \longrightarrow \begin{array}{ccc} & \uparrow & \\ \leftarrow & & \rightarrow \\ & \downarrow & \end{array} \quad \boxed{\mathrm{Y}} \quad \boxed{\mathrm{Z}}$$

where $\boxed{\mathrm{X}}$ is specified by the last applied rule, while the direction and new labels $\boxed{\mathrm{Y}}$ and $\boxed{\mathrm{Z}}$ are specified in the current move. In our encoding

$$\begin{array}{l} 0 \longrightarrow \boxed{\mathrm{A}} \\ 1 \longrightarrow \boxed{\mathrm{B}} \\ 2 \longrightarrow \boxed{\mathrm{C}} \\ 3 \longrightarrow \boxed{\mathrm{D}} \end{array}$$

and similarly for directions

$$\begin{array}{l} 0 \longrightarrow \downarrow \\ 1 \longrightarrow \uparrow \\ 2 \longrightarrow \rightarrow \\ 3 \longrightarrow \leftarrow \end{array}$$

For example the string

$$0, 2, 3, 1, 1, 2, \ldots$$

specifies the following moves: For initialization, we ignore the first two numbers which specify a direction and a label and start off with 3 which specifies the second label, which represents $\boxed{\mathrm{D}}$. The next two numbers tell us to go up (1) after replacing the previous label ($\boxed{\mathrm{D}}$) by the label denoted by 1 which is $\boxed{\mathrm{B}}$. Once we have replaced the previous label we go up (as already indicated) and label the cell $\boxed{\mathrm{C}}$ as indicated by the number 2. We continue in this way for a total of nine moves. The result of the first move is shown below.

$$\boxed{\mathrm{D}} \quad \Rightarrow \quad \begin{array}{c} \boxed{\mathrm{C}} \\ \boxed{\mathrm{B}} \end{array}$$

With this encoding, crossover always results in a legal string and allows the generation of new grammars and application sequences. Thus, over time, the algorithm learns improved grammars and searches for application sequences leading to the generation and exploration of new design spaces. Figure 9 shows a part of a typical genotype and the resulting structure or phenotype.

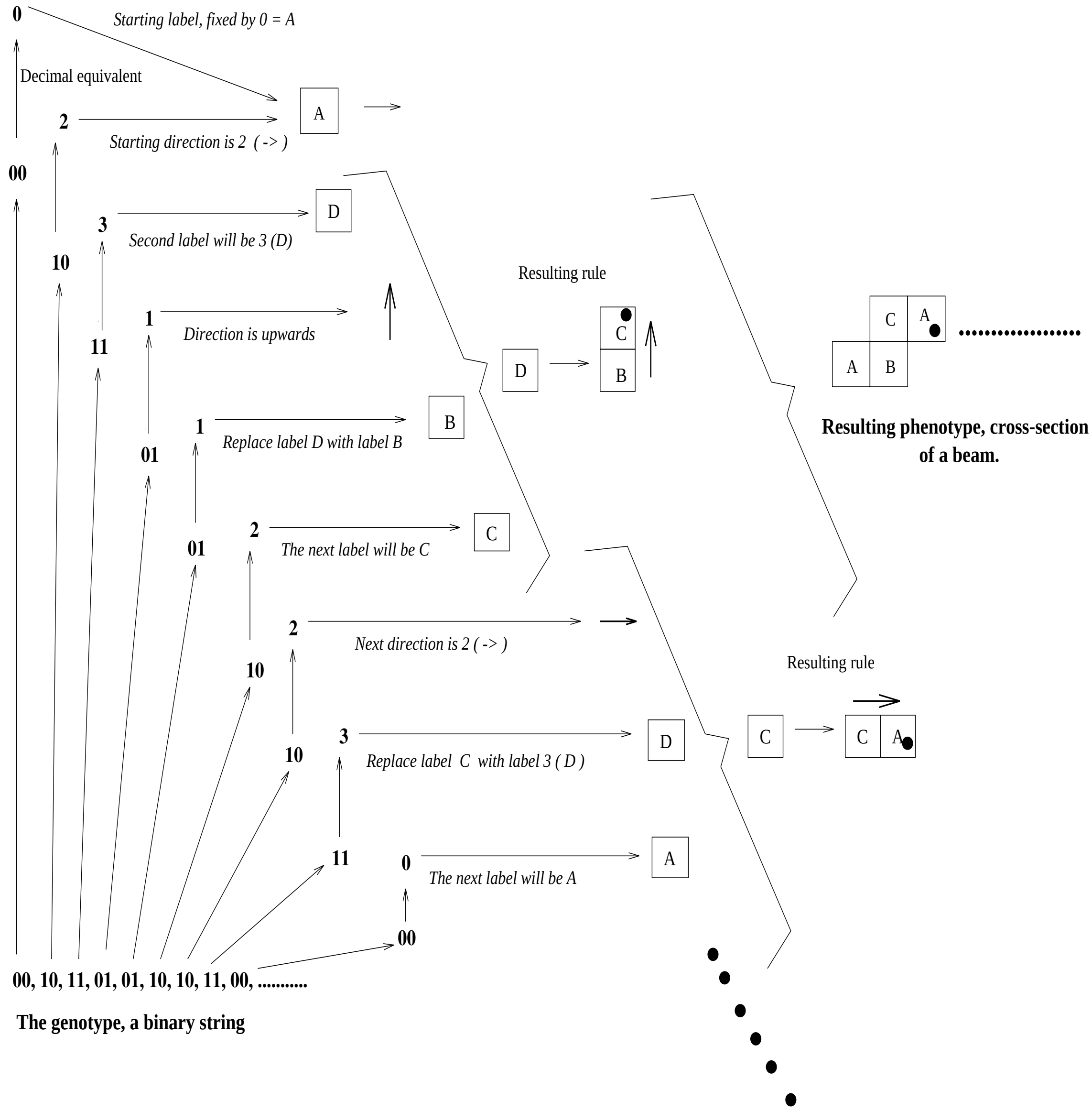

Figure 9: A typical genotype string and its resulting structure.

As indicated earlier, we define a two criteria fitness function for the genetic algorithm. Individuals for mating are chosen randomly and recombined. The two parents and two children form the set of solutions from which we extract the Pareto optimal set using these criteria through exhaustive generate and test. Those individuals that belong in the Pareto optimal set participate in the next generation of the the genetic algorithm. The GA completes a generation when repeated application of the above procedure fills up the fixed size population of 50. The population size of 50 was chosen experimentally as sizes larger than that had very little effect on the performance of the system. The initial population is generated randomly. We use the Pareto optimal set and its inverse (Radford and Gero 1988), over 11 runs of the GA with different random seeds to compare results and solutions. The space described by the Pareto set and its inverse is called the *space of feasible solutions.* Each run of the GA terminates after 100 generations/iterations. Results from a number of runs are usually used in comparative studies of genetic algorithm applications since the algorithm is probabilistic in nature and a single run may not provide a true comparison. 11 runs provides a sufficiently large sample for our experiments. 100 generations was experimentally determined to be enough for the GA to converge. The population size was chosen based on the computing resources available and the need to have sufficient diversity at initialization (see Louis and Rawlins (1992) for details on population sizing and convergence time).

4.3 Results

Figure 10 shows the feasible solution spaces for our fixed grammar and the learned grammar. The two feasible spaces are not equal and the space depicting feasible solutions for the learned grammar shows a marked improvement in performances. Let S_0 denote the behavior space for the structures defined by our fixed grammar, and S_n the space when the grammar is allowed to evolve. Figure 10 indicates that:

$$S_0 \not\subset S_n$$
and
$$S_0 \bigcap S_n \neq \phi$$

This corresponds to a substitutive process. All genetic algorithm parameters were the same for the runs with the fixed grammar and the runs with the grammar that is allowed to evolve. The reason for the difference and improvement follows from the observation that the heaviest label D is now allowed to propagate in the vertical direction, thus increasing the moment of inertia. Since perimeter does not depend on weight, but only on shape, the genetic algorithm learns grammars that increasingly use D labeled cells, leading to larger values for the moment of inertia. Figure 11 shows how the genetic algorithm exploits cells labeled D by increasing their utilization at the expense of the other labelled cells. Figure 12 shows a grammar that was learned by this evolutionary process. This grammar is capable of producing better designs than the commencing grammar. Figure 13 shows some of the beam sections produced during the learning process. These could not be produced with the original grammar.

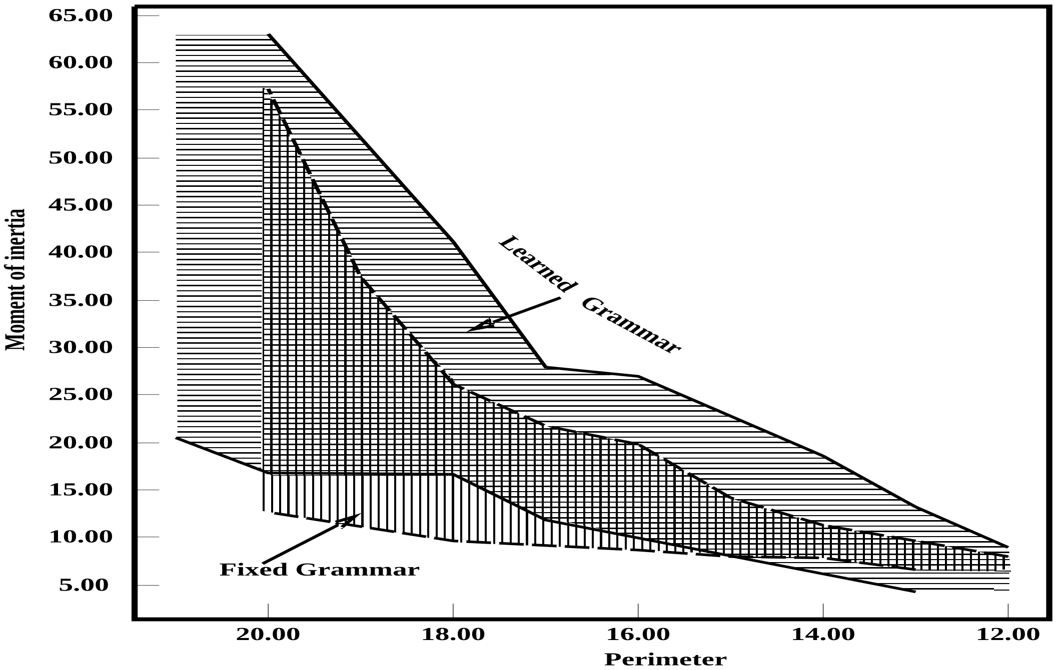

Figure 10: The spaces of feasible solutions produced when using the fixed grammar and when allowing the system to learn a new grammar.

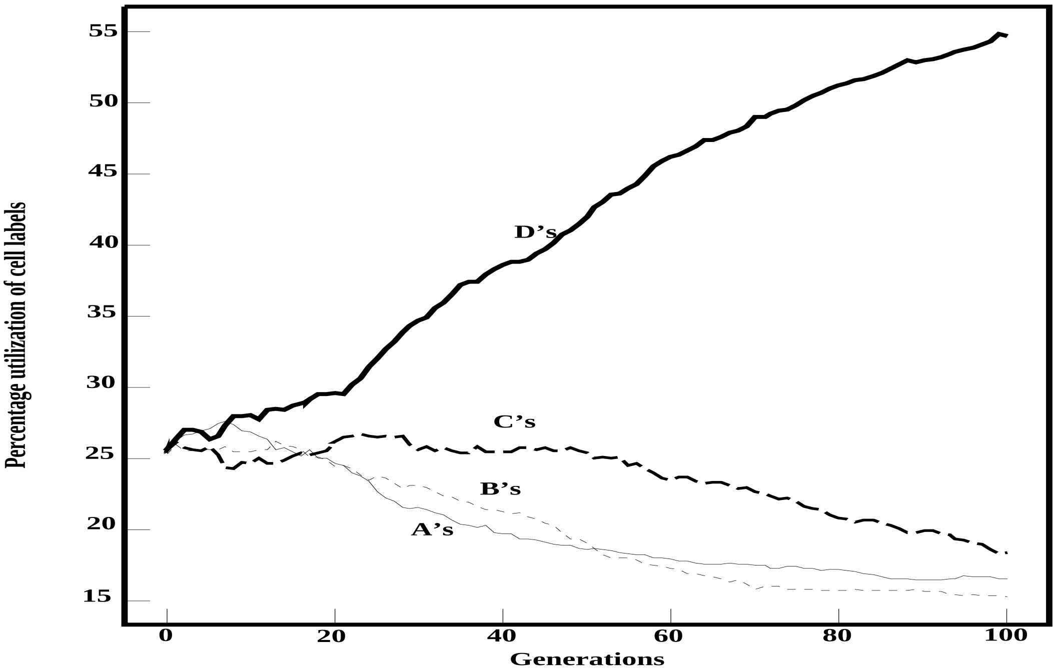

Figure 11: The number of cells labeled D increases with time (generations) while the numbers of the other cells decrease as the system learns new grammars.

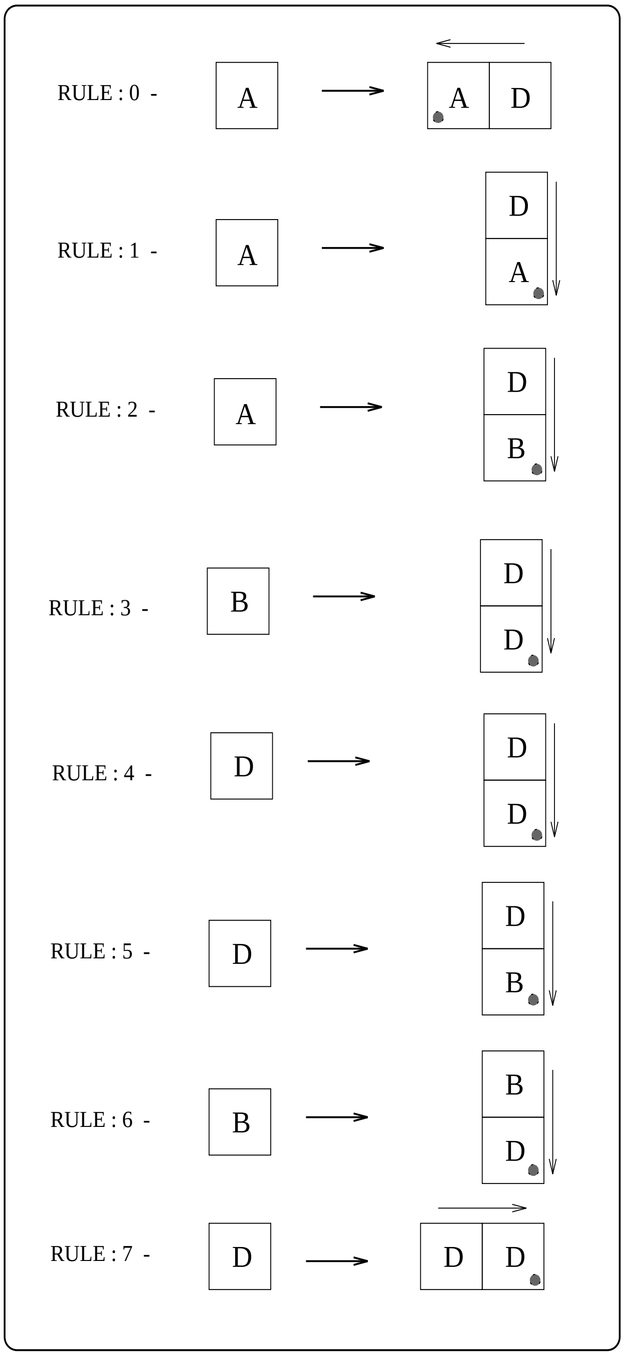

Figure 12: A learned shape grammar.

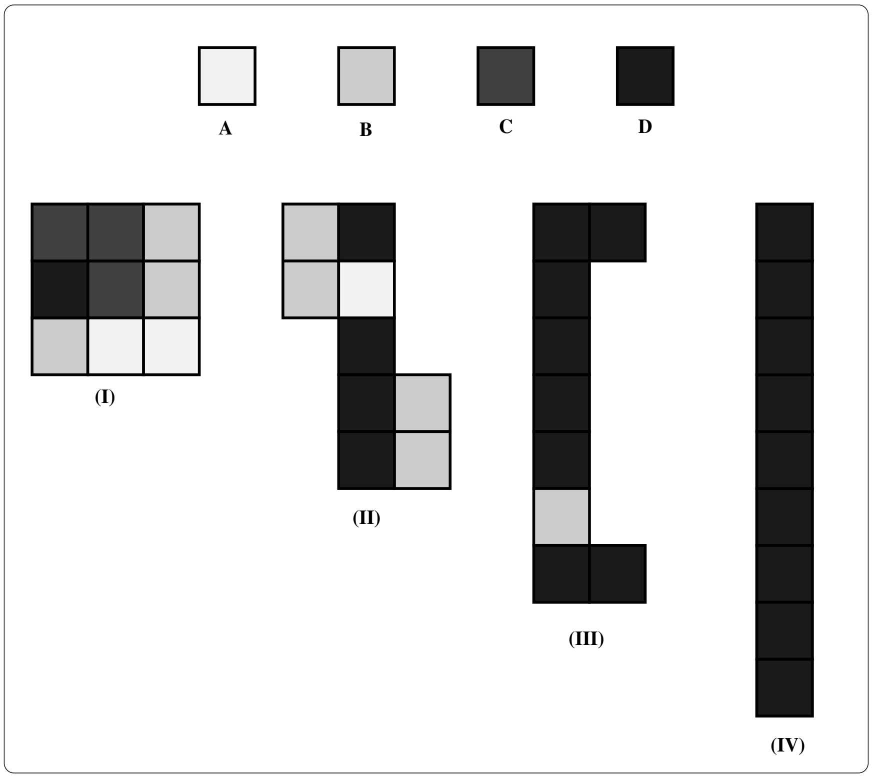

Figure 13: Some beam sections produced during the learning process.

5 Discussion

Much of machine learning in design is focussed on learning to perform old tasks more efficiently. The emphasis is on generalizing from examples. In design this generalization forms the basis of routine design. Its application results in designs of the same kind as the examples. Its utility is founded on one of three bases:

- Complex casual theoretical models which are computationally expensive are replaced by more abstract phenomenological models which are computationally inexpensive.

- Weak casual models are replaced by strong phenomenological models which are founded on cases (examples).

- Where no casual models exist the phenomenological model provides a supportable computational approach.

All of these work well within a conception of design as problem solving and search. The generalizations provide alternative approaches to searching (or generating) the states in the state space.

However, we wish to treat design not simply as the processing of searching amongst pre-existing solutions, ie, within a fixed state space, but as a process of search and exploration via learning. By exploration we mean the creation of alternative state spaces that can then be searched to produce more useful solutions. What we have demonstrated is that, the genetic algorithm formulation can provide a computational construct to carry out the learning process. We used the function-behavior-structure + knowledge framework of design prototypes and utilized the shape grammar formalism as the genetic basis of the genetic algorithm. Two aspects of shape grammars were encoded at the genotype level: the order of execution of the rules of the shape grammar and the rules themselves. This provided the opportunity for both the order of execution and the rules themselves to change. The former manifests itself as a means of searching a fixed state space for a given set of rules in the grammar. The latter manifests itself as a means of exploration by creating new state spaces through the evolution of new rules. Such a system learns new grammars through the use of an evolutionary process underpinning both search and exploration.

In the example we commenced with a simple eight rule shape grammar and limited its use to nine rule applications. We then let the system learn new shape grammars which improved the design of a beam section for moment of inertia and for perimeter. The system demontrated substitutive characteristics as exemplified in Figure 10 where the state spaces of behaviors produced by the learned grammars both overlapped and was partially disjoint with the state spaces of behaviors produced by the original grammar.

This form of exploration uses an evolutionary learning process. It demonstrates that learning need not be case based in design; that learning can occur at a more abstract level than from cases and that learning can form the basis of exploration in design.

6 Acknowledgements

This work has been supported by an Australian Research Council grant to John Gero in the area of memory-based non-routine design. Some of the supporting ideas were initially presented at the AID'92 Workshop on Search-Based and Explanation-Based Models of Design and at the AID'92 Workshop on Machine Learning in Design. Both these workshops formed part of the Second International Conference on Artificial Intelligence in Design '92.

References

[1] Arciszewski, T. , Mustafa, Z. and Ziarko, W. (1987). A methodology of design knowledge acquisition for use in learning expert systems, *Man-Machine Studies* **27**: 23-32.

[2] Carbonell, J. G. (1990). Introduction: paradigms for machine learning. *in* Carbonell, J. (ed.) *Machine Learning Paradigms and Methods*, MIT/Elsevier, Cambridge, Massachusetts, pp. 1-10.

[3] Gero, J. S. (1987). Prototypes: a new schema for knowledge based design, *Working Paper,* Architectural Computing Unit, Department of Architectural Science, University of Sydney, Sydney.

[4] Gero, J. S. (1990). Design Prototypes: a knowledge representation schema for design, *AI Magazine* **11**(4): 26-36.

[5] Gero, J. S. (1992). Creativity, emergence and evolution in design. *in* Gero, J.S. and Sudweeks, F. (eds), *Preprints : Second International Round-Table Conference on Computational Models of Creative Design*, Department of Architectural and Design Science, University of Sydney, pp. 1-28.

[6] Gero, J. S. and Kumar, B. (1993). Expanding design spaces through new design variables, *Design Studies* **14**(2): 210-221.

[7] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Massachusetts.

[8] Gunaratnam, D. S. and Gero, J. S. (1993). Neural network learning in structural engineering applications, *in* L. F. Cohen (ed.), *Computing in Civil and Building Engineering, Vol. 2*, ASCE, New York, pp. 1448-1455.

[9] Holland, J. (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

[10] Louis, S. J. and Rawlins, G. J. E. (1992) Syntactic analysis of convergence in genetic algorithms *in* D. Whitley (ed.), *Foundations of Genetic Algorithms 2*, Morgan Kauffman, San Mateo, CA, pp. 141-152.

[11] Mackenzie, C. A. and Gero, J. S. (1987). Learning design rules from decisions and performances, *Artificial Intelligence in Engineering* **2** (1): 2-10.

[12] Mackenzie, C. A. (1989). Inferring relational design grammars, *Environment and Planning B: Planning and Design* **16**: 252-287.

[13] Mackenzie, C. A. (1991). *Function and Structure Relationships and Transformations in Design Processes*, PhD Thesis, Department of Architectural and Design Science, University of Sydney, Sydney.

[14] Maher, M. L. and Li. H. (1992). Automatically learning preliminary design knowledge from design examples, *Microcomputers in Civil Engineering* **7**: 73-80.

[15] Maher, M. L. and Li, H. (1993). Adapting conceptual clustering for preliminary structural design, *in* L. F. Cohen (ed.), *Computing in Civil and Building Engineering, Vol. 2*, ASCE, New York, pp. 1432-1439.

[16] McLaughlin, S. and Gero, J. S. (1987). Learning from characterised designs, *in* D. Sriram and R. Adey (eds), *Artificial Intelligence in Engineering: Tools and Techniques*, CM Publications, Southampton, pp. 347-359.

[17] Mitchell, T. M. , Carbonell, J. G. and Michalski, R. S. (eds) (1986). *Machine Learning : A Guide to Current Research*, Kluwer, Boston.

[18] Quinlan, J. R. (1979). Discovering rules by induction from a large collection of examples, *in* D. Michie (ed.), *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, Edinburgh, pp. 168-201.

[19] Quinlan, J. R. (1986). Induction of decision trees, *Machine Learning* **1**: 81-106.

[20] Radford, A.D. and Gero, J.S. (1988). *Design By Optimization in Architecture, Building, and Construction*, Van Nostrand Reinhold, New York.

[21] Rao, R. , Lu, S. , and Stepp, R. (1991). Knowledge-based equation discovery in engineering domains in machine learning, *in* L. Birnbaum and G. Collins (eds), *Proc. Eighth International Workshop in Machine Learning (ML 91)*, California.

[22] Reich, Y. (1991). Design knowledge acquisition: task analysis and a partial implementation, *Knowledge Acquisition* **3**: 237-254.

[23] Stiny, G. and Gips, J. (1978). *Algorithmic Aesthetics: Computer Models for Criticism and Design in the Arts*, Unversity of California Press, Berkeley and Los Angeles, California.