Brock University

Department of Computer Science

**Enabling and Measuring Complexity in Evolving Designs
using Generative Representations for Artificial Architecture**

Adrian Harrington
Technical Report # CS-12-09
October 2012

Enabling and Measuring Complexity in Evolving Designs using Generative Representations for Artificial Architecture

by

Adrian Harrington

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Brock University, St. Catharines, Ontario

Abstract

As the complexity of evolutionary design problems grow, so too must the quality of solutions scale to that complexity. In this research, we develop a genetic programming system with individuals encoded as tree-based generative representations to address scalability. This system is capable of multi-objective evaluation using a ranked sum scoring strategy. We examine Hornby's features and measures of modularity, reuse and hierarchy in evolutionary design problems. Experiments are carried out, using the system to generate three-dimensional forms, and analyses of feature characteristics such as modularity, reuse and hierarchy were performed. This work expands on that of Hornby's, by examining a new and more difficult problem domain. The results from these experiments show that individuals encoded with those three features performed best overall. It is also seen, that the measures of complexity conform to the results of Hornby. Moving forward with only this best performing encoding, the system was applied to the generation of three-dimensional external building architecture. One objective considered was passive solar performance, in which the system was challenged with generating forms that optimize exposure to the Sun. The results from these and other experiments satisfied the requirements. The system was shown to scale well to the architectural problems studied.

Acknowledgements

I would like to thank the following people who helped me in all aspects of my research and the preparation of this thesis:

- Brian Ross for his inspiration, guidance and being so easygoing.

- Cara Vickers and my parents, Donna and Alan for all their support.

- Steve Bergen for sharing all his work with me.

- Cale Fairchild for all his linux and cluster computing expertise.

- Brock University and the Computer Science Department for all their resources.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Evolutionary design (ED) is an area of evolutionary computation, in which we apply evolutionary algorithms to evolve art, designs or other creative solutions. This area is of particular interest when discussing the evolution of complexity because we are not necessarily seeking an 'optimal' solution when evolving designs or art. We are however searching for unique or innovative results which help to inspire designers and artists. Typically, it is solutions which have some complexity that we find interesting. Since the introduction of ED, we have sought after enhancing the scalability of systems to evolve solutions to problems of greater complexity. While there have been many examples of human-competitive results created by evolutionary design systems, the quality of swolutions that can be evolved must be further improved to obtain wider acceptance in academic and industrial communities.

Before pushing the boundaries of what can be accomplished in evolutionary design, we must decide on what types of characteristics and attributes enable us to solve these more difficult problems. We also require metrics for measuring how varying characteristics and attributes in evolutionary systems enable better solutions to more difficult problems. There is some existing research which tries to tackle improving upon complexity in evolutionary systems. Hornby argues that modularity, reuse and hierarchy are characteristics of designs to complex problems and are present in solutions in both engineering and software development [17]. The attributes which enable these three characteristics, he argues, are combination, control-flow and abstraction [15]. He develops an evolutionary system which enables these three attributes and creates metrics which attempt to monitor how each of them is present in a candidate solution. His findings suggest that the best solutions are found when all these characteristics are enabled. This is a rich problem area which requires more investigation and backing through implementation and analysis.

In an effort to achieve the generation of higher quality objects, many researchers are examining the evolution of grammars, which are then used to create solutions or families of solutions [16, 32].

Using this technique, the creation of robust phenotypes is made possible through the evolution of simple genotypes. The use of L-systems, a specialized form of grammar, has also become quite popular. It is of interest how these grammar systems compare, in both achieving high fitness scores and enabling better solutions for problems of increased complexity. It is also interesting to consider what features of the languages help the evolutionary system in obtaining better solutions.

Artificial architecture is a new and emerging area of research which marries the two fields of artificial intelligence and architectural design. Architects are challenged with the difficult task of designing entire building structures, which requires the use of computer-aided design software and many other tools. There is, however, a lack of tools which help with the creative process in their design phases [45]. Applications could be developed which aid in the creative process by generating results automatically. This seems like an appropriate problem for an evolutionary design system. Research exists in the automatic generation of external building architecture [7], though could be re-examined using the principles and characteristics of complexity to obtain less abstract and more practical structures.

It is apparent that further research is required in gaining a better understanding of what principles and characteristics play a role in developing better solutions in ED problems, as well as measuring the scalability of complexity that they enable. We wish to analyze how varying rules of grammars and L-systems help to enable better designs and how these principles will affect the generation of evolutionary designs, such as the generation of external building architectures.

A GP system has been developed, extending the popular ECJ system. The system implements individuals as tree-based generative representations. These generative representations are modeled very closely after Hornby's work. They are similar to L-systems, though they have access to parameters, conditionals and iterative bodies. The system is capable of automatically generating three-dimensional form, meeting evaluation criteria to fill space and examine Sun exposure. Examples of what can be produced with the developed system are shown in Figures 1.1 and 1.2.

1.1 Goals

There are three major sets of goals to be achieved in this research, all of which are described in detail here. The first is to examine the performance and validity of generative representations in evolutionary design problems. The second is to examine a problem in the domain of architectural design, an area of research that has shown great interest in evolutionary computation. The third goal is to examine design languages and operators for three-dimensional form generation.
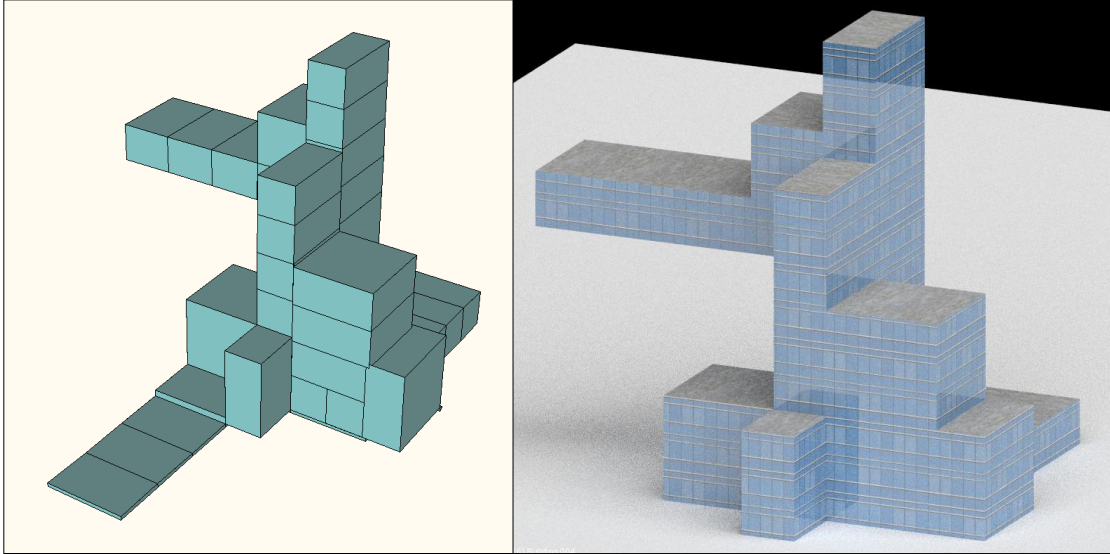
Figure 1.1: Generated model from Sun exposure experiment (left). Texture and shading applied as a post-processing effect (right).
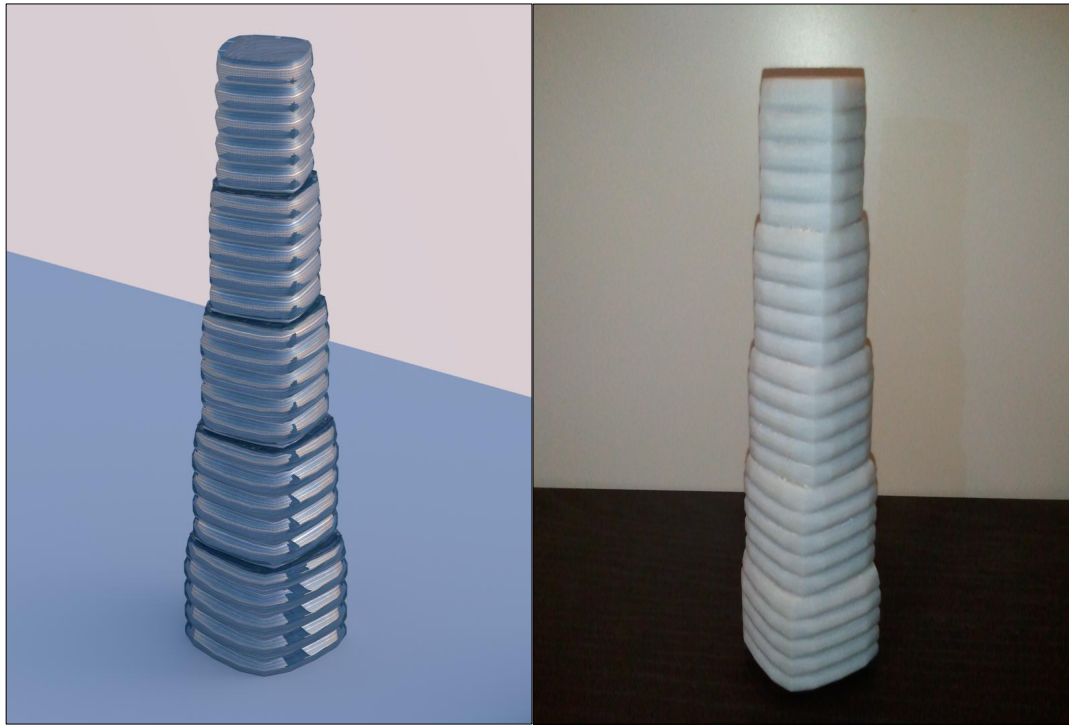
Figure 1.2: Generated model from Skyscraper experiment. Textured and shaded (left). 3D print (right).

1.1.1 Representation and Complexity in Evolutionary Design

A primary goal is to observe how principles and measures of complexity help to improve the quality of solutions in evolutionary design problems. A GP system will be developed which encodes individuals as tree-based generative representations. These generative representations will be based on Hornby's work, and we will examine them in a new problem setting, measuring his characteristics of complexity to determine their affect on the quality of solutions [17]. Comparing factors such as recursion, iteration, named procedures, parameterization and control-flow in representations will also be performed.

To justify that the principles and characteristics of complexity play a significant role in evolutionary design, the following will be performed:

- Generative representations, inspired by Hornby, will be implemented and applied in a new problem setting.

- Characteristics of complexity will be measured and compared to determine how they improve the quality of solutions, and in what way they contribute improvements.

- Problem and representational constraints will be investigated, to determine their ability in limiting complexity in design constructs.

1.1.2 Automatic Generation of Architectural Form

The secondary goal is to apply these improved representations to the area of artificial architecture. We will create a system capable of automatically generating external building architectures through an evolutionary process, providing bio-inspirational form. This work will further extend Corrado and Ross's research [7], to create more feasible three-dimensional architectural designs.

To achieve the creation of an evolutionary design system which automatically generates architectural form, the following will be considered:

- The development and implementation of a set of evaluation functions for filling space with three-dimensional form.

- Experimention with external factors, such as light and shadow.

- Comparing the sensibility and feasibility of designs generated by the generative & non-generative representations.

1.1.3 Design Languages for Form Generation

The third goal is to experiment with various design languages and operators for the generation of three-dimensional form. Design objects will be generated in an open-space as opposed to a

voxelized-space to allow for limitless possibilities. These languages will be designed by combining various low-level operators and experimented on, in conjunction with the generative representations.

Through this research, the following goals related to form generation are to be achieved:

- The creation of single, water-tight, open-space design objects, using an evolutionary system with generative representations.

- Determining which languages perform well in a three-dimensional design problem.

- Gain some understanding of generated styles and aesthetics, especially pertaining to architectural synthesis.

- The development of a set of scalable design problems.

1.2   Thesis Structure

This thesis is structured a follows. Chapter 2 provides relevant background information, providing a brief summary of evolutionary computation, genetic programming, multi-objective optimization and generative grammars. Chapter 3 reviews literature inspirational to this work and relevant to the fields of evolutionary design, generative representations, complexity, form generation and artificial architecture. Chapter 4 provides details of the GP system that was designed and developed, and describes how generative representations were implemented and interpreted for the creation of design objects. Chapter 5 summarizes the various design languages used to generate form and the methods used for evaluating design objects.

Chapter 6 and 7 introduce the first set of experiments that attempt to compare generative representations and their performance in a simple and difficult design problem. Chapter 8 extends experimentation, examining design languages and external factors such as lighting in form generation. Chapter 9 contains a set of miscellaneous experiments that were undertaken.

Chapter 10 summarizes the work performed in this thesis and provides suggestions for future work.

Chapter 2

Background Information

This chapter introduces the fundamental concepts that drive the proposed system and research. It provides a brief and general overview on the subjects of evolutionary computation, genetic programming, multi-objective optimization, and generative grammars including shape grammars & L-systems. Readers who are familiar with these subjects can proceed to Chapter 3.

2.1 Evolutionary Computation and Genetic Programming

Evolutionary Computation (EC) is a methodology inspired by Darwinian evolution, wherein a population of individuals is evolved to meet some evaluation criteria, following the principles of natural selection and survival of the fittest [21, 35]. It is a technique that allows a computer to automatically solve problems, based on a high-level statement of what needs to be achieved. While there are many varying techniques in the field of EC, genetic programming (GP) is the focus of this research. GP differs from other methodologies by treating each individual of the population as its own computer program, capable of executing within some problem domain. GP is a stochastic process, and does not guarantee optimal solutions or to completely solve problems. However, it has proven to be a practical approach for creating acceptable solutions to many complex and time consuming problems.

2.1.1 Evolutionary Algorithm

Genetic Programming [21] and Genetic Algorithms [12] are both classified as evolutionary algorithms. These algorithms simulate the passing of time as a set of step-by-step events which are called generations. The process begins by initializing a population of either randomly generated or pre-existing individuals. Once initialization is complete, the evolutionary phase begins. Individuals are evaluated at each generation on how well they solve a problem according to some fitness criteria. Individuals which are deemed more fit are selected for reproduction with other individuals. This

breeding operation creates new individuals with characteristics of their parents, enabling the system to create combinations of various solutions, searching through the domain of potential solutions. This process of evaluation and swapping of genetic material repeats for many generations, with the goal of improving the population's evaluation score over time. Eventually concluding by some termination criterion, individuals with the best evaluation scores will surface as potentially practical solutions to a given problem. An overview of the algorithm is provided in a step-by-step manner in Figure 2.1 and is discussed in the following sections.

---

1. Generate a random or seeded initial population of GP trees using available primitives.

2. Evaluate initial population using selected fitness function(s).

3. From 1 to *max_generations* or until another stopping criteria is met, repeat the following steps:

   (a) Select two individuals from population with priority based on individual fitness.

   (b) Perform crossover on the couple with probability P_c, producing two children.

   (c) Mutate offspring with probability P_m.

   (d) Repeat steps (a – c) to create a new population of children.

   (e) Replace the old population with the new and evaluate fitness of each individual.

4. Return the best individual or set of individuals found.

Figure 2.1: Overview of GP algorithm.

2.1.2 Individual Representation

Genetic programming works similarly to genetic algorithms and other techniques in EC. One of the main distinctions of GP is the representation of individuals - sometimes called chromosomes - in the population. Individuals are themselves executable computer programs, typically represented as syntax trees of operators and terminals. These trees are taken and executed, typically reading each node in a depth-first manner. An example of a mathematical function represented as a tree structure and its evaluation are presented in Figure 2.2. This example individual is composed of mathematical operators, the variable x and some constant values as the potential operators and terminals. The set of operators and terminals are unique to each problem and many possibilities and combinations can exist. They are typically designed by those implementing the GP system to tackle a certain problem and are often referred to as a GP language.

As problems become more complex, GP can be extended to allow the ability to constrain types on its operators and parameters. In this way, a parent node will only have children which make sense in that context. Having the ability to constrain program trees by types allows for better expressiveness

and structure. Strongly typed GP enforces that every function, argument and terminal has a return-type [29].

Evaluation:
(5 / cos(x)) * (x + 3)
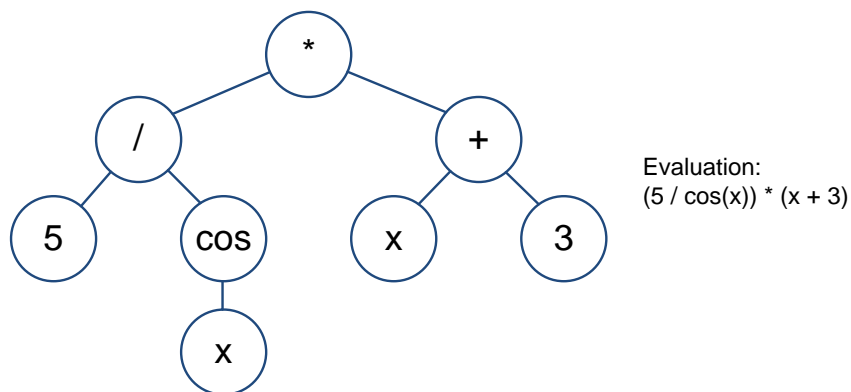
Figure 2.2: Example GP individual and evaluation.

2.1.3   Initialization

Each GP run begins with an initialization phase to create a population of individuals. Emphasis has been put into creating efficient and effective initialization strategies [21]. To initialize a population completely randomly would result in the generation of many similar tree structures and low population diversity. An effective initialization method would focus on creating trees of varying structures. One particularly useful strategy is the half-and-half initialization. In this strategy half of the generated individuals are created as balanced trees and the other half are created as non-balanced trees. There are also many other parameters, which control minimum and maximum depths. In this manner the population has a diverse set of tree structures. It is also possible to seed a population with pre-fabricated individuals.

2.1.4   Evaluation

At the beginning of every generation, each individual in the population is evaluated. The score that it receives during evaluation is its fitness. The fitness function for each GP system is problem dependent and needs to be designed as such. The fitness score is a quantitative measure of how good or bad an individual is at solving a specific problem. Individuals with better fitness scores have increased probability of being selected at each generation for the creation of new individuals through reproduction.

2.1.5   Selection

After all of the individuals in the population have been evaluated, the reproduction process to create the next generation's population begins. Two members from the population need to be selected for reproduction. These two members are called parents. There are various techniques for selecting parents for cross breeding. To simply pick the overall best individuals at each generation would result in early convergence of the population to a particular solution. A popular choice is the tournament selection strategy which randomly selects k individuals from the population and takes the best two for reproduction [35]. The selection process is repeated for as many times as reproduction needs to take place.

2.1.6   Reproduction Operators

Reproduction operators exist to generate new combinations of individuals. There are two types of reproduction operators: crossover and mutation. Crossover takes two individuals in the population using a selection strategy and creates two new individuals by exchanging sub-trees. Mutation creates new individuals by selecting a node in the individual's syntax tree and randomly generates a new sub-tree. An example of crossover is shown in Figure 2.3 and an example of mutation is shown in Figure 2.4.
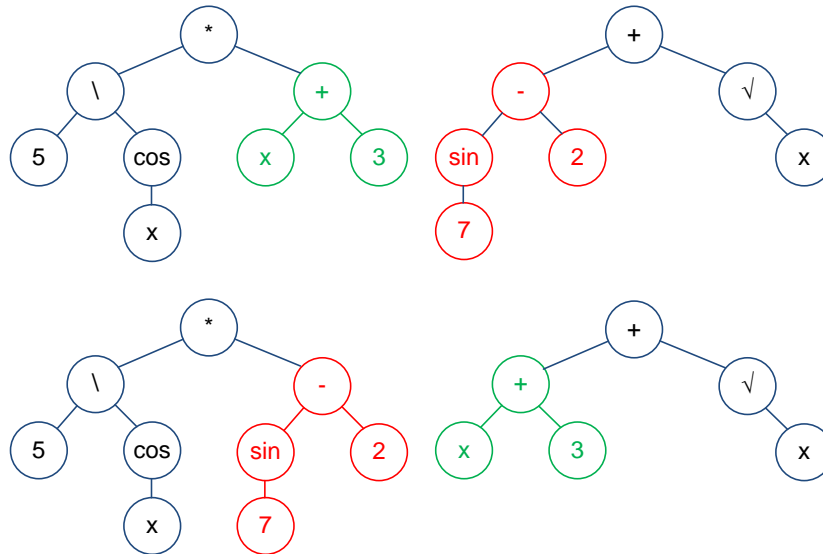
Figure 2.3: Example of Crossover. Two parents (top) swap sub-trees, creating two children (bottom).
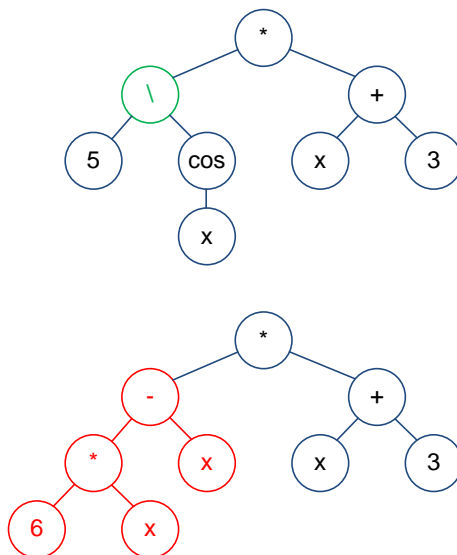
Figure 2.4: Example of Mutation. Node is selected for mutation and a new random sub-tree is generated.

2.1.7   Elitism

Due to the somewhat destructive nature of the reproduction operators, elitism can be introduced to help preserve the best individuals in the population during evolution. Elitism helps defend against losing the best individuals by preserving them in the population throughout the run. This works by saving the best k individuals at the beginning of a generation, then re-injecting them into the new population after reproduction operations have been applied. Elitism can also potentially have negative effects, forcing the population to prematurely converge, causing stagnation of genetic diversity in the population.

2.1.8   Automatically Defined Functions

Early in its lifespan, it was recognized that GP could benefit from the three-stepped problem solving process that most humans take when examining complex problems. This solving process involves first decomposing the problem into sub-problems, then solving these smaller and simpler problems, then reassembling these smaller solutions into a larger and final solution. Koza presents a mechanism for performing this task called automatically defined functions (ADF) [22]. ADFs allow genetic programming to define dynamic sub-routines created from the GP language, which are contained in sub-trees. These sub-routines are then called from the individual's main tree. This technique promotes the reuse of smaller programs and has shown great success.

2.2 Multi-objective Optimization

Many problems exist that require two or more objectives to be optimized simultaneously in a single evolutionary run. The simplest method is to aggregate multiple fitness scores together or apply a weighted-sum of all features in the fitness vector. This adds bias however; and also presents difficulties in choosing appropriate weights. Often these objectives can be conflicting and it sometimes does not make logical sense in the context of the problem to aggregate objectives together. As opposed to single-objective optimization problems which generally only require one optimal solution, multi-objective problems can have many potentially good solutions, which optimize the problem features in different ways.

Multi-objective systems will often generate entire sets of solutions, where ideal candidate solutions excel in most or all of the fitness objectives. Sometimes generated solutions are very good in one feature domain and poor in the others. These candidates are considered outliers and are undesirable.

Extensive research has taken place in multi-objective optimization techniques [6]. Pareto ranking is a classic optimization technique and is explained below. Summed ranking is an extension of Pareto ranking and is the strategy used in this research. Table 2.1 presents an example of how a given set of fitness vectors are transformed into the various ranks systems.

2.2.1 Pareto Ranking

Pareto ranking is a multi-objective ranking strategy which attempts to maximize the performance of multiple objectives concurrently without combining them into a single fitness [6]. During evaluation, each individual is assigned a fitness vector comprising of a score for each feature being examined. Ranking is then performed by examing which individuals are *dominated* by others. An individual A dominates B if it is at least equal in all objectives and better in at least one. Let V be a feature vector who is made up as such: $\overrightarrow{V} = (v_1, ..., v_n)$. then the following is true in a maximization problem

$$A \ dominates \ B \ iff \ \exists i : a_i > b_i \wedge \forall i : a_i \geq b_i \tag{2.1}$$

Pareto ranks are assigned to each individual in passes, where each individual is compared to all other members of the population. In the first pass, all individuals that are un-dominated are assigned a rank of *1* and removed from the current working population. The current rank is incremented by one and the following pass is performed on all remaining members of the current working population, assigning each un-dominated individual the current rank. This process is repeated for each subsequent pass until all members of the population have been assigned a rank. These ranks are then used as the fitness score during the evolutionary process.

2.2.2   Normalized Rank-Sum

Normalized Rank-Sum extends the Pareto ranking strategy, attempting to eliminate outliers often generated by Pareto and improve its performance in higher dimensionality [1].

It works by first transforming an individual's feature vector $\overrightarrow{V} = (v_1, ..., v_n)$ into a rank vector $\overrightarrow{R} = (r_1, ..., r_n)$. For each feature v_i in an individual's feature vector, it is ranked against the rest of the population's v_i and is given a rank r_i. A normalized rank n_i is then calculated by retrieving the maximum rank r_k for that feature and performing r_i/r_k . The final fitness score is obtained as $fitness = \sum_i n_i$.

Table 2.1: Examples of ranking calculations using Pareto and Rank-Sum strategies.

Fitness Vector	Pareto Rank	Rank Vector	Sum	Rank-Sum	Normalized Sum	Normalized Rank-sum
(25, 60, 40)	1	(2, 1, 2)	5	1	1.3	1
(36, 99, 31)	1	(3, 2, 1)	6	2	1.8	2
(42, 60, 84)	2	(4, 1, 3)	8	4	1.9	4
(91, 99, 65)	2	(5, 2, 4)	11	5	2.8	6
(91, 60, 31)	1	(5, 1, 1)	7	3	1.7	3
(13, 99, 92)	1	(1, 2, 5)	8	4	2.2	5

2.3   Generative Grammars

Generative grammars were first introduced by Chomsky as a means for generating and processing languages in a formal manner, using a set of rules [5]. A grammar can be generalized as containing an alphabet of symbols and a series of rewrite rules which alter those symbols to create new strings of symbols. Grammars have proven to be very useful in a wide variety of applications and research areas. Shape grammars and L-systems are two specialized extensions of the generative grammar and are explained in further detail here.

2.3.1   Shape Grammars

Proposed by Stiny, shape grammars are a systematic method for creating geometric shapes and patterns, based on the principles of a generative grammar [42]. A shape grammar is typically composed of an alphabet of two and three-dimensional shapes, as well as a set of rewriting rules that perform changes on those shapes. The rewriting rules can be carried out serially or in parallel, allowing for the exploration of new shapes and designs in an iterative fashion. A formal definition of a shape grammar is as follows [42]:

A shape grammar SG is a 4-tuple: $SG = (H_T, H_M, R, I)$ where,

1. H_T is a finite set of shapes.

2. H_M is a finite set of shapes such that $H_T^* \cap H_M = \emptyset$.

3. R is a finite set of ordered pairs (u, v) such that u is a shape consisting of an element of H_T^* combined with an element of H_M and v is a shape consisting of:

   (a) the element of H_T^* contained in u or

   (b) the element of H_T^* contained in u combined with an element of H_M or

   (c) the element of H_T^* contained in u combined with an additional element of H_T^* and an element of H_M.

4. I is the initial shape consisting of elements of H_T^* and H_M.

Figure 2.5 provides a simple example of the shape exploration process, showing a derivation tree with a depth of two. Beginning with the initial shape, each potential rewrite rule is applied to generate a new branch of the derivation tree. Designs generated by a system will belong to the same language, therefore looking similar in some manner. If one were to treat shape exploration as a search problem, then a shape grammar will limit that search space by only allowing the generation of new objects within the constraints of the grammar system. In this way, shape grammars are used as a method for exploring an inexhaustible number of designs, created by shapes [43].
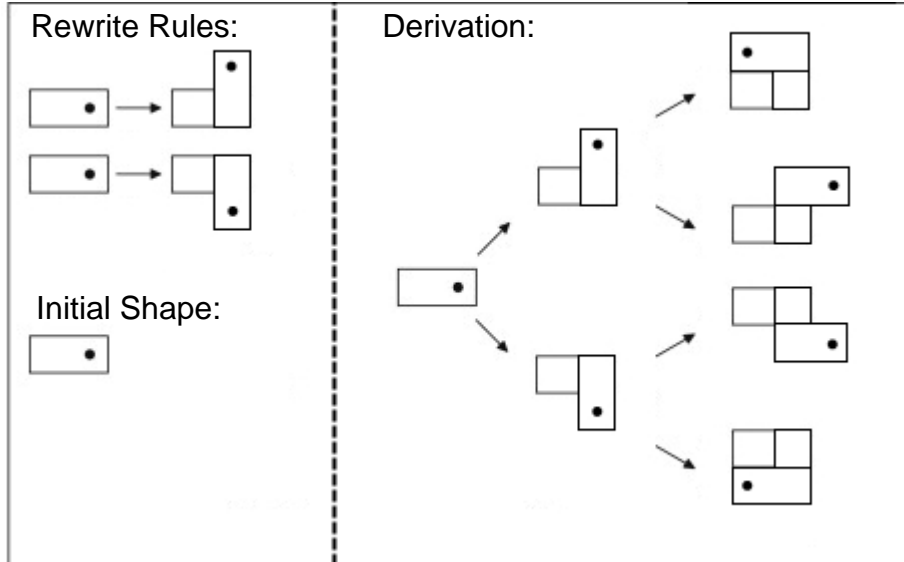
Figure 2.5: Example of shape grammar derivation tree.

The use of shape grammars has become quite popular in the field of computation and computer-aided architectural design. This is due to the fact that grammars are formal languages which can be

represented, constructed and executed on a computer. As a formal method for representing designs, they have gained ground in evolutionary computation and evolutionary design in both two and three-dimensional problem settings. By automatically evolving grammars with genetic programming or other evolutionary techniques, designers are able to develop new, inventive and inspirational designs for design problems.

2.3.2 L-systems

The Lindenmayer system (L-system) is another variant of the formal grammar which was introduced to model the biological development of multicellular organisms [23]. The distinguishing feature of the L-system lies in the application of rules. Beginning with the initial state, each rule that can be applied is carried out in parallel in a left-to-right fashion. This process results in the creation of only one new object at each rewrite depth. The resulting language is in fact simply a subset of the language a formal grammar would generate. This allows for a much more manageable result set, especially when creating systems for use in evolutionary computation. For example, the following L-system:

$$a: \rightarrow ab$$
$$b: \rightarrow a$$

when started with the initial symbol a, will produce the following strings at each rewriting step:

$$a$$
$$ab$$
$$aba$$
$$abaab$$
$$abaababa$$
$$abaababaabaab$$

2.3.3 Parametric L-systems

The parametric L-system (PL-system) further extends the L-system [24]. It allows for the passing of parameters during rewrite calls, algebraic expressions can be applied to parameters and conditionals can determine branching of rewrite rules. A single production rule now consists of three parts: the predecessor, the condition and the successor. The predecessor is only rewritten to the successor if the condition passes. For example, the following PL-system:

$$a(x): (x > 3) \rightarrow a(x/2)b(2)$$
$$a(x): (x \leq 3) \rightarrow a(x+1)$$
$$b(x): (x > 1) \rightarrow a(0)b(x-2)$$
$$b(x): (x \leq 1) \rightarrow b(x+1)$$

when started with *a(6)*, will produce the following strings at each rewriting step:

$$a(6)$$
$$a(3)b(2)$$
$$a(4)a(0)b(1)$$
$$a(2)b(2)a(1)b(2)$$
$$a(3)a(0)b(0)a(2)a(0)b(0)$$

Chapter 3

Literature Review

This chapter provides a review of the fields of research relevant to this thesis. It contains a brief overview of evolutionary design, in-depth analysis of Hornby's generative representations and measures of complexity, and a look at what is being done in the fields of three-dimensional shape evolution and artificial architecture.

3.1 Evolutionary Design

Evolutionary computation has been applied to many fascinating problems in the field of art and design, creating a relatively new and intriguing area of research called evolutionary design. In fact, it is currently an active area of research [20], with entire textbooks devoted to the subject [8, 36].

There have been many novel applications of evolutionary algorithms which highlight its power in creating solutions for a wide variety of design problems. Among these applications include the ability to track a criminal suspect through Face-Space [4], or the automatic generation of music [28]. In this section, we will examine what is current in the field of evolutionary design.

Sims is among the first people to examine evolution for computer graphics, creating populations of images based on evolved expressions [39]. This helped to pave the way for all sorts of artistic applications. Later, he worked on a system that evolved autonomous three-dimensional virtual creatures [40]. There are a number of compelling examples of evolved two-dimensional images, including images generated from assemblages of three-dimensional objects [27]. Evolutionary algorithms have since been applied to the generation of abstract and three-dimensional form for both applications in design and art.

The use of generative grammars in evolutionary design systems has garnered some popularity. Shape grammars present opportunities to aid in design processes and early design support systems, specifically in spatial arrangements and limiting design space to speed up the process of route

selection through problem space [41]. O'Neil examines the use of shape grammars with grammatical evolution to rediscover two-dimensional target structures, and evolve logo designs [31, 32]. Jacob combines genetic programming with L-systems for the evolution of artificial flowers, generating realistic results [19]. Bergen improves on the combination of GP with L-systems to automatically generate three-dimensional structures with features of aesthetics [2].

Here, we examined just a small subset of the work being done in this field. There is still a vast amount of research and exploration to be done.

3.2 Generative Representations and their Performance Metrics

In most genetic programming systems, a population of individuals are represented as syntax trees. These syntax trees are parsed and applied directly towards solving a problem, where some fitness criteria is measured. There is a concern with this method however, that it does not scale well to problems of growing complexity.

Generative representations are a different approach to the encoding of individuals in genetic programming [16]. In this technique, syntax trees create some form of grammar which is then parsed to generate the solution. In Hornby's work, the generative representations are encoded as L-systems, which allow for abstraction, control flow and combination.

In subsequent research, Hornby measures the characteristics of modularity, reuse and hierarchy in his representations to contrast how they relate to fitness scores [17]. Here, modularity refers to the grouping of elements so that they can be manipulated as a unit. Reuse is a repetition or reuse of elements in a design. Hierarchy refers to the amount of layers of modules and elements in a structured design.

Representations are created with these various features enabled and disabled on a table design problem. In these experiments, interesting results are discovered which are worth noting here, due to the relevance in this work. First and foremost, he discovers that generative representations outperform non-generative representations in this problem domain. Furthermore he finds that higher fitness is achieved in representations with more features of modularity, reuse and hierarchy enabled. Five sets of experiments were performed, which compared different representations, enabling the varying principles: modularity (M), reuse (R), hierarchy (H). These five sets were MRH, MH, MR, M and none. It was shown that MRH outdid the others and that the system with none enabled performed the worst overall. MR and MH obtained relatively the same fitness scores and both outperformed the representation with just M enabled.

In his trials, he examines the performance of systems on varying sizes of the table design problem (10x10x10, 20x20x20, 50x50x50). In each of these experiments, the generative representation with

the principles of MRH enabled outperforms all others. What is important to note is that the amount by which it outperforms the non-generative representation grows for each larger problem size. This shows that the generative representation enables scalability for solving larger problems. Finally he shows the relevance of the metrics of complexity in scatter diagrams, which compare fitness to each of the measures of modularity, reuse and hierarchy. It can be generalized that fitness is shown to be better in representations where each of these characteristics are more present. Also, solutions that have higher scores in each of these metrics tend to have better average fitness. As this technique is showing great promise, we wish to further prove its worth.

In the pursuit of developing high quality solutions to complex problems using evolutionary systems, one must consider which characteristics and attributes enable the system to achieve these solutions. Of course, it is difficult to determine which characteristics enable better solutions, without measuring and understanding how these characteristics improve the solution. Therefore, it is important in this research to determine metrics for examining how these characteristics aid in creating more scalable evolutionary systems.

As mentioned in the problem statement, generative representations with the characteristics of modularity, reuse and hierarchy have been shown to generate better solutions for complicated problems in Hornby's work [18]. To gain some perspective on how these three characteristics play a role in allowing evolutionary design systems to perform better in more complex problem domains, metrics were created that examine individuals in both genotype and phenotypic form. A number of metrics used in Hornby's research are listed below in Table 3.1, some of which will be examined when developing a system which generates external building architecture.

3.3   Artificial Architecture

Artificial architecture, synonymous with algorithmic design and algorithmic architecture, is the application of artificial intelligence to architectural design. Architects have been using computers to aid them with their designs for many years. Computers can automate well defined and preconceived processes. Terzidis claims that there is a lack of software which attempts to explore the processes which we can't quite put our fingers on [44]. This is an active area of research amongst architects, who have become increasingly interested in the power of evolutionary computation for its ability to explore creative designs [45]. A call for systems which aid in the creative design process for generating structural form has been made [45], to alleviate problems such as design fixation, by providing a range of inspirational models.

While there are some general rules and ideals, there is no exact science for determining which architectural forms will be pleasing to people [38]. Artificial architecture is a method through which we attempt to generate space and form through rule-based logic [44]. There are many goals and evaluation methods to consider when developing architectural form. An example of such a goal

Table 3.1: Measures of complexity.

Measure	Description
Modularity	The number of times a structural module is contained in a design.
Reuse	The average number of times elements are used to create a design.
Hierarchy	The number of layered modules in a design.
Algorithmic Information Content	The algorithmic information content of a string is the length of the shortest program which can reproduce that string.
Design Size	The opposite of the algorithmic information content, it is the size of what is produced by the design program.
Logical Depth	The number of symbols processed in converting the genotype to the phenotype.
Sophistication	The number of control symbols produced (loops, conditional statements, procedure calls).
Number of Build Symbols	A count of the non-control symbols.
Grammar Size	The number of production rules generated in the design.
Connectivity	The maximum number of edges that can be eliminated, before the graph is split into two parts.
Number of Branches	A count of nodes that have two or more children.
Height	The height of the generated tree.

is to maximize the amount of natural light that hits a form. Watanabe tackled this in his Sun God City project, which built form through rule-based logic [46]. Algorithmic techniques are also an important aspect of this problem space. Terzidis examines methods for generating architecture through algorithmic means, proposing various strategies for form generation [44].

Work has been performed on the procedural synthesis of architectures, using only generative grammars. Shape grammars have been used in creating plans for whole cities [13, 34], procedurally modelling a cityscape in 3D, including buildings, roads and plant life. They have also been used to procedurally model and generate buildings [30], including structurally-sound masonry buildings [47]. Systems have also been created which allow for interactive editing of grammars for the generation of architecture [25]. The drawback of generative grammars is that they must all be handcrafted on a problem by problem basis.

The specific area of research we are interested in is the application of evolutionary design systems for architectural problems. Creating tools powered by these techniques would enable the automatic generation of a variety of styles and novel designs. GENR8 is a design tool, developed using a grammatical evolutionary for the generation of organic surface [33, 14]. The evolution of the system is guided by target points and user interactivity. Frazer evolves building envelopes and towers using mathematical functions to generate form [10, 11].

O'Neill *et al.* have applied genetic programming with shape grammars to the creation of three-dimensional shelters [32]. A novel method for automatically evolving three-dimensional building topologies has been developed [7]. Here they use genetic programming to evolve shape grammars,

which in turn render building architectures. These shape grammars are then interpreted by a commercial application called City Engine.

Through cooperation of algorithmic techniques and the human mind, we should be able to achieve higher quality solutions to complex design problems. The field of artificial architecture is beginning to redefine architectural practice [37].

Chapter 4

System Details

This chapter contains in-depth details of the system that was developed for use in this research. It describes all of the implementation details concerning the genetic programming system and generative representation encodings. It provides diagrams and examples of how a GP-tree is represented and how it is transformed into a design encoding.

4.1   Overview

To achieve the goals presented in Chapter 1, a system was designed and developed with the following objectives in mind:

- GP individuals are required to be encoded as generative or non-generative representations.

- Complexity metrics must be implemented and calculated at run time.

- The system will be focused on the automatic generation of diverse and novel three-dimensional models.

- The system must work with various design languages and operators.

- The evaluation system should allow for multi-objective problems and interchangeable fitness functions.

- Parameters and constraints must be easy to input and alter.

The problem was tackled by developing a genetic programming system which extends ECJ, allowing for generative representations and the requirements listed above. ECJ is a flexible GP package developed using Java [26]. The system also implements the JNetic framework [3], an interactive UI which displays design artifacts during the evolutionary process and provides tools for the selection and loading of experimental parameters. Genetic programming was selected as the appropriate

methodology as it is well suited for the creation of grammars due to its tree-like nature. An extensive amount of effort was placed into extending those systems to allow for the encoding of generative representations as a tree structure. Furthermore, the implementation of complexity metrics and a module for decoding representations into design encodings was carried out as well. The final product is a flexible system that meets all of the goals outlined above.

4.2 System Execution

The series of steps taken during the execution of the system are overviewed in Figure 4.1. The execution of the evolutionary run begins by allowing the user to select parameters from a UI, or by loading them from a parameter file. These include generational algorithm parameters, such as population size, maximum generations, reproduction options, etc.. The selection of a generative representation is then required to use as an encoding. This encoding is enforced by the GP language which is loaded from a function set file. The user must then select a design language composed of operators which are used to build the design objects at run-time. Finally, the user selects one-to-many fitness functions, which are evaluated using the normalized rank-sum evaluation strategy.
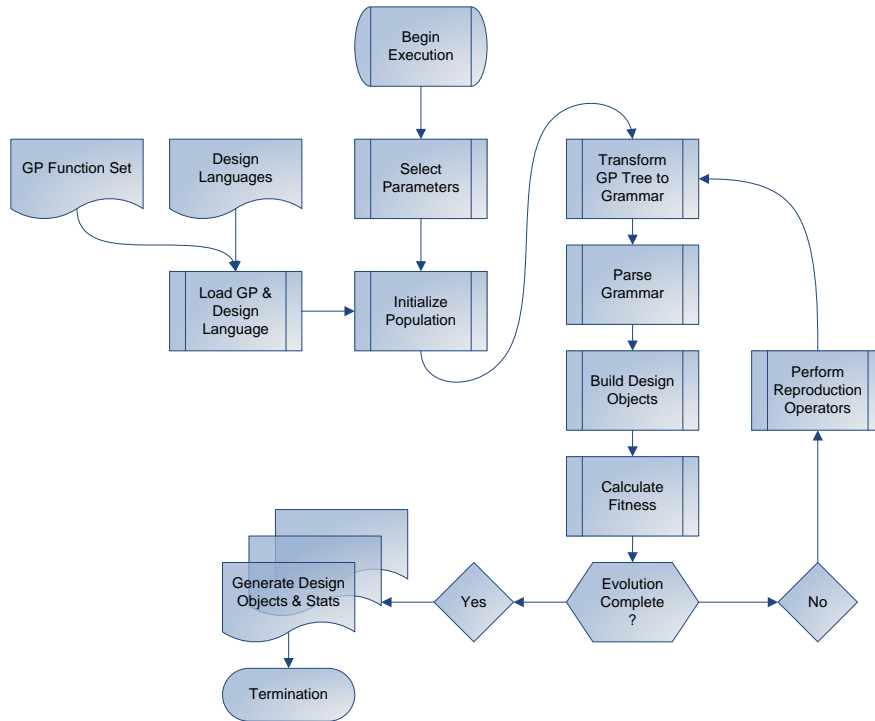
Figure 4.1: System Execution.

Once all parameters and languages are selected, the evolutionary process is kicked off by generating a randomly created initial population. The generational process then takes place, repeating six main steps. The first step transforms each GP individual from their tree encoding into a generative grammar. The generative grammar is then parsed, creating a design encoding. Each individual then generates a three-dimensional design object from their design encoding. Fitness calculations and complexity measures take place on the design objects and grammars. Once fitness is calculated, the system decides if the evolutionary process is complete. If not complete, reproduction operators are applied, creating a new population of individuals. This process is repeated for each generation.

When the number of generations has reached the limit, the evolutionary process comes to an end. The final population of design objects are output as individual Wavefront .obj, text & image files. Statistics pertaining to evolutionary performance and complexity measures are also created in Excel format for experimental analysis.

4.2.1 Diversity

The creation of diverse population sets is extremely important in the domain of evolutionary design. To enforce this, a diversity strategy created by Flack was implemented to penalize potential duplicate individuals [9]. The strategy works by examining the fitness vector of each individual per generation. If two individuals are evaluated to the same fitness scores, they are deemed to be clones of one another and one individual is given a penalty score. It is important to note that it is possible that two individuals could potentially differ as individuals but achieve the same fitness scores. This fact is disregarded when using this strategy, as it is an uncommon event in this problem domain.

4.3 Generative Representations

In a typical GP system, each individual of the population is generated as the solution, which is then directly applied to a problem. When using generative representations, each individual is created as a grammar which in turn is used to generate a design encoding. In this manner, individuals are enabled to achieve higher forms of complexity through attributes such as abstraction, modularity, and control-flow.

Table 4.1 maintains a list of all the parameters associated with generative representations in this system. Five encodings are implemented in this system with various combinations of modularity, reuse and hierarchy enabled. These representations are:

Simple: No features are enabled in this representation. It contains only one procedure, whose condition always evaluates to true. The RHS of that procedure is limited to the *DesignProcedure-Limit*. This representation is similar to the classical GP encoding.

Modular: In this encoding, procedures and procedure calling are allowed. Only the first procedure, *A()*, is allowed to execute any other procedure. The *ProcAmount* parameter determines how many procedures will be created and the *ProcConditionAmount* is limited to one.

Modular-Reuse: This representation enables repetitions as well as procedures. It enforces all of the constraints of the Modular encoding, though *ProcConditionAmount* can be set to more than one. *RepMin* and *RepMax* must also be set and allow for constraining the repetitions amounts.

Modular-Hierarchical: Hierarchy is enabled by allowing procedures to call one another, creating a stack of procedure calls. Here, each procedure is limited by *ProcMaxUse*, enforcing a maximum on the amount of times a procedure may be called.

Modular-Reuse-Hierarchical: This encoding enables all of the features. It allows for multiple procedures, procedures to call any other procedures and the use of repetitions. *RewriteMin* and *RewriteMax* determine the minimum and maximum rewrite depths allowed be each system.

Table 4.1: Generative Representation Parameters

Parameter	Description
Encoding	
Encoding	The encoding to be used.
	Options include Simple, M, MR, MH & MRH.
Re-write Procedures	
ProcAmount	The amount of procedures that will be created.
ProcArity	The amount of arguments that will be contained in each procedure.
ProcConditionAmount	The amount of Condition-Pairs per procedure.
ProcMaxUse	The maximum amount of times a procedure may be called.
Language Symbols	
SymbolArity	The amount of arguments in a symbol call.
ERC Limitations	
RewriteMin	The minimum rewrite depth.
RewriteMax	The maximum rewrite depth
RepMin	The minimum repetition amount.
RepMax	The maximum repetition amount.
ConstantMin	The minimum constant integer value.
ConstantMax	The maximum constant integer value.
String Limitations	
EncodingLimit	A string size limit on the encoding.
RHSLimit	The string size limit on a RHS of a production rule.
DesignProcedureLimit	The string size limit of the entire design procedure.
	(Symbols exceeding the length are cropped.)

4.4 GP Language

The GP Language in this system is created to enable the selection of either a non-generative or one of the four generative representations. It is a strongly-typed language, enforcing each individual to maintain a particular structure. Table 4.2 contains descriptions for each of the types in the system and its general hierarchy. The types used by the system include: root, procedure, list, character and integer types. The root type is required as the parent node for each GP-tree, enforcing structure. The procedure types are used to generate and maintain production rules. List, character and integer types are used in the creation of design strings.

Table 4.2: GP types used in the language

Representation	Description
Atomic Types	
Root	Parent node that maintains the structure of the GP tree.
P	A rewrite procedure of the generative system.
L	A list of characters.
C	A single character.
I	An integer type.
Subtypes of P	
P_{rule}	A production rule.
P_{cond}	A Condition-Pair.
Subtypes of L	
L_{list}	A list containing a series of symbols and procedures.
L_{RHS}	The right hand side of a production rule.
L_{arg}	An argument of a procedure or symbol call.
Subtypes of C	
C_{proc}	An upper-case character representing a procedure.
C_{symbol}	A character representing a design symbol.
C_{param}	A lower-case character representing a parameter.
Subtypes of I	
$I_{rewrite}$	Representation of the amount of times the system will be re-written.
I_{reps}	This type determines how many times a repetition block will occur.
I_{val}	An integer containing some ERC value.

Table 4.3 references the function set that the GP system has access to when creating individuals. Each tree is created with a root node called *GenRep*, that maintains nodes which determine the rewrite depth, starting string parameter values and one-to-many production rules. There are two variants of the production rule type: *CondRule & SimpleRule*. The simple rule has no condition and the RHS of the rule always fires. The conditional rule has one-to-many *CondPair* nodes. Each condition-pair node maintains a parameter to examine and a constant to compare it to, to determine if the procedure will be rewritten to the RHS.

The RHS node is the body of the rewrite rule. It fathers three list nodes to aid in the speedy

creation of longer strings. Here, many list functions are used, including procedure calls, symbol calls, list branching, push-pops, repetition and mathematical operators. Functions for the creation of integer and character terminals are also implemented. Figure 4.2 provides an example of the general layout of a single GP individual. Nodes containing an asterisk would contain more details but are omitted due to space.
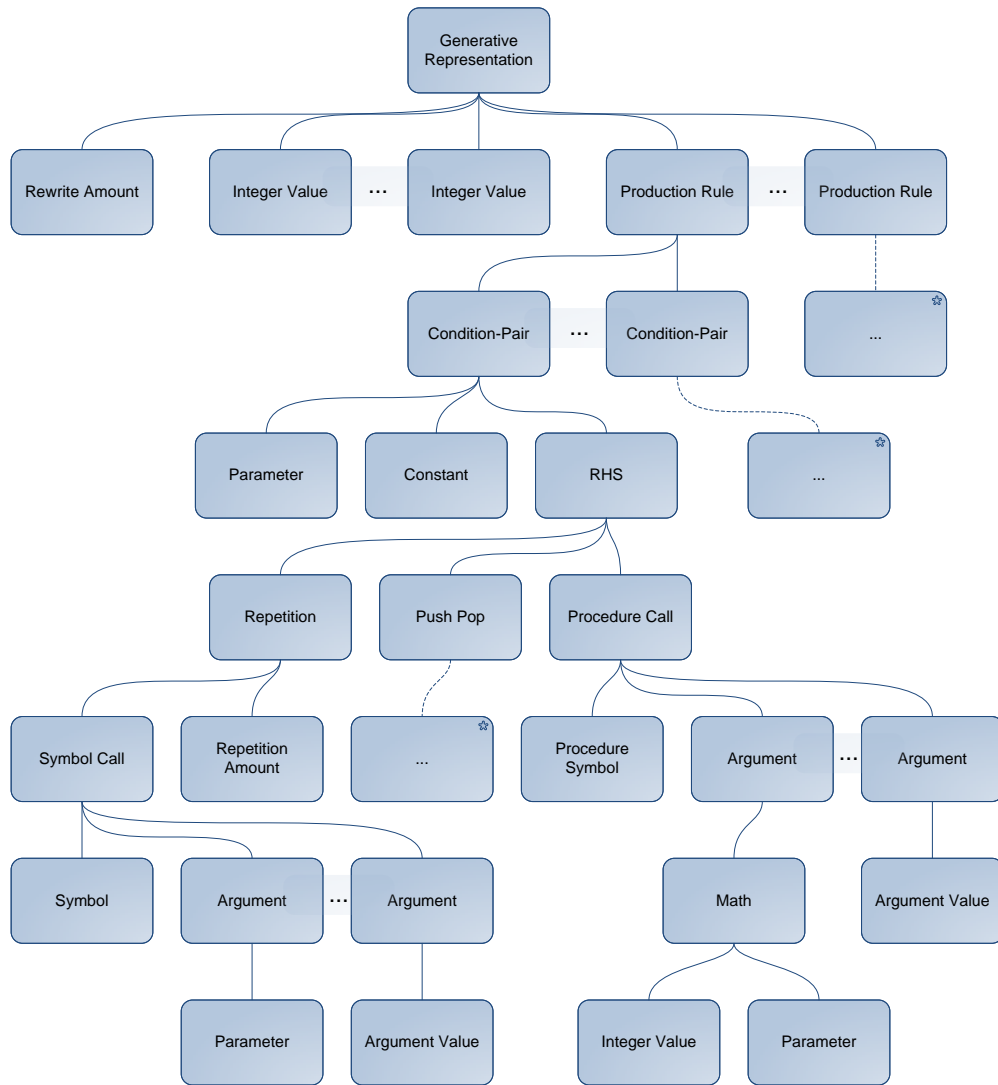
Figure 4.2: Generative GP tree representation.

Table 4.3: Function set for the GP Language

Returns	Function	Description
Root	GenRep(I$_{rewrite}$, I$_{val}$, P$_{rule}$)	Returns a complete generative encoding, where the number of I$_{val}$ is the ProcedureArity and the number of P$_{rule}$ is the ProcedureAmount.

Returns	Function	Description
P$_{rule}$	CondRule(P$_{cond}$)	A production rule, containing one-to-many condition-production pairs where the amount is determined by ProcedureConditionAmount.
P$_{rule}$	SimpleRule(L$_{RHS}$)	A simple rule that contains no condition, only a RHS.
P$_{cond}$	CondPair(C$_{param}$, I$_{val}$, L$_{RHS}$)	A single condition-pair containing a parameter, integer value and RHS.

Returns	Function	Description
L$_{RHS}$	rhs(L$_{list}$, L$_{list}$, L$_{list}$)	RHS of a production rule.
L$_{list}$	ProcCall(C$_{proc}$, L$_{arg}$)	A procedure call that contains arguments, where the number of arguments is the ProcedureArity.
L$_{list}$	SymbolCall(C$_{symbol}$, L$_{arg}$)	A symbol call that contains arguments, where the number of arguments is the SymbolArity.
L$_{list}$	ListBranch(L$_{list}$, L$_{list}$)	A list composed of two other lists, allowing branching.
L$_{list}$	PushPop(L$_{list}$)	A list wrapped by [and].
L$_{list}$	Repetition(L$_{list}$, I$_{reps}$)	A list wrapped by '{' and '}(I$_{reps}$)'.
L$_{list}$	ListStopper( )	Returns an empty list, to stop infinite tree growth in a Simple encoding.
L$_{arg}$	ArgumentValue( )	Returns an integer value between ConstantMin and ConstantMax.
L$_{arg}$	ArgumentString( )	Returns a parameter symbol as an argument.
L$_{arg}$	Math(I$_{val}$, C$_{parameter}$)	Returns some mathematical operation on a parameter and value.
L$_{arg}$	Math2(C$_{param}$, C$_{param}$)	Returns some mathematical operation on two parameters.

Returns	Function	Description
I$_{val}$	intTerminal( )	An integer ERC within the range of ConstantMin and ConstantMax.
I$_{reps}$	repTerminal( )	An integer ERC within the range of RepMin and RepMax.
I$_{rewrite}$	rewriteTerminal( )	An integer ERC within the range of RewriteMin and RewriteMax.

Returns	Function	Description
C$_{proc}$	procedure( )	A single procedure character ERC.
C$_{param}$	parameter( )	A parameter character ERC.
C$_{symbol}$	symbol( )	A design symbol character ERC.

4.5   Example of Representation Transformation

An example of an MRH encoded GP-tree with sample values is shown in Figure 4.3. A walkthrough of the transformation from GP-tree to design object is presented here. In this very simplified example the *ProcAmount* is set to 1, *ProcArity* to 2, *ProcConditionAmount* to 1, and *SymbolArity* to 1.

The transformation begins by examining the children of the *Generative Representation* node who are not production rules. The rewrite amount is determined from the ERC value and the starting string is constructed by the integer value nodes. The starting string will always call the first production rule *A()*, where the amount of arguments is set by the *ProcArity*. The starting string and rewrite depth from this example are interpreted as follows:

Starting string: *A(1,3)*
Rewrite depth: *3*

Next, the production rules are determined:

A → b > 4: rt(1) [ ] A(3,a)
A → b > 1: {ex(a)}(2) [ ] A(b,4+a)

Once the generative system has been created, it is rewritten as follows:

Starting string: *A(1,3)*
Rewrite 1: *{ex(1)}(2) [ ] A(3,5)*
Rewrite 2: *{ex(1)}(2) [ ] rt(1) A(3,3)*
Rewrite 3: *{ex(2)}(2) [ ] rt(1) {ex(3)}(2) [ ] A(b,4+a)*
Design encoding: *ex(2)ex(2) rt(1) ex(3)ex(3)*

The design encoding is then taken and executed to generate a three-dimensional model. In this context, *ex()* refers to extruding along the active face and *rt()* changes the active face to the current face's right neighbour. Figure 4.4 displays the generated model.
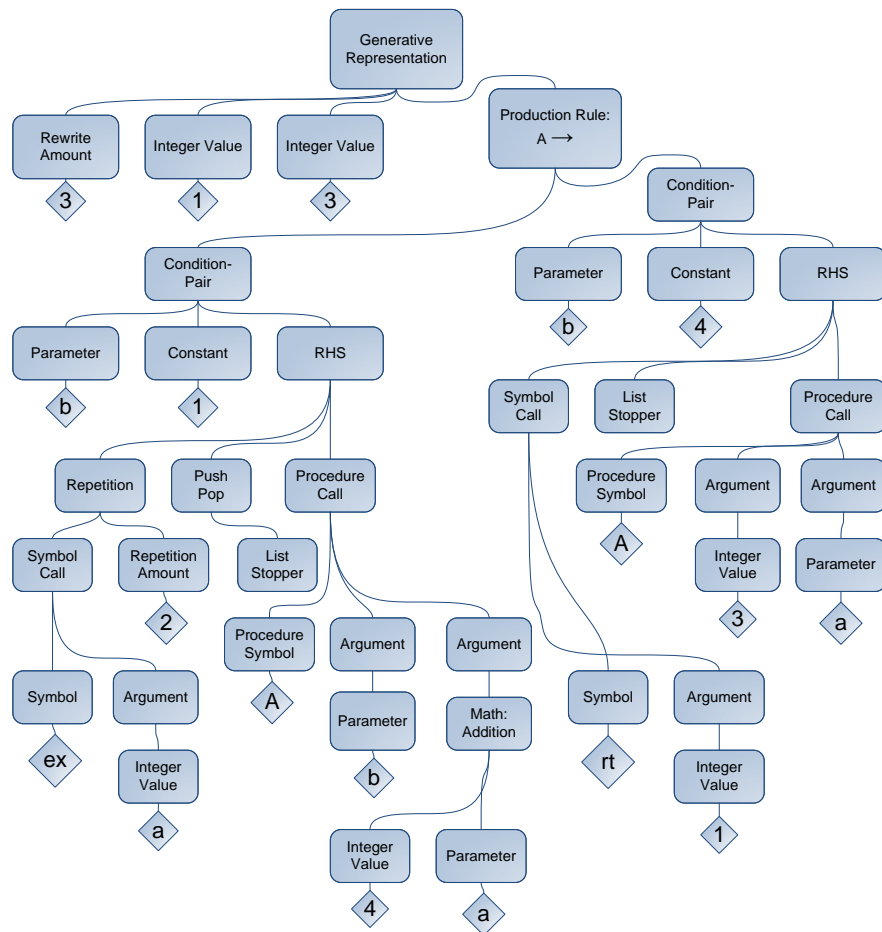
Figure 4.3: Sample GP-tree individual, using the MRH representation.
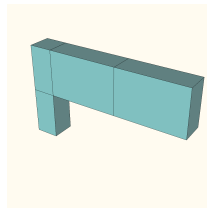
Figure 4.4: Model generated from walkthrough.

Chapter 5

Design Languages and Evaluation

This chapter contains an overview of the design models to be evolved and their properties. It provides information of the various design languages and operators that were implemented for the synthesis of design models. It also shares details on the fitness functions used to evaluate these models.

5.1   Overview of Generated Artifacts

The goal of this research is to utilize an evolutionary system for the automatic generation of three-dimensional design artifacts. We wish for the creation of diverse sets of architectural form, which are compelling, novel and inspirational. The creation of these models will be directed by both the design languages that are available to the system and the types of fitness functions that will guide the evolution of those models.

The architectural designs generated by the system will be both fully connected and watertight. A fully connected model is one that does not have any disjoint components. Watertight is a term used to describe if a model is suitable for 3D printing. This means that there are no holes or missing faces which expose the interior of the model. These two properties enforce that the model has some structural integrity and will allow for the models to be rendered with a 3D printer. An example of a building envelope generated by the system is shown in Figure 5.1.

Typically, the generation of three-dimensional form in evolutionary design is done using either only surface-based techniques or in a voxelized space. In this research, we wish to generate free form, voluminous models, with minimal constraints. The benefits of using polygonal form include being able to evolve more novel artifacts with many unique surface-normals, removing the need for post-processing steps which can bias the emerging design process.

Examining a non-voxelized space presents a series of extra difficulties. A lack of strict boundaries means that a system can evolve an artifact which is extremely large, complex and time consuming
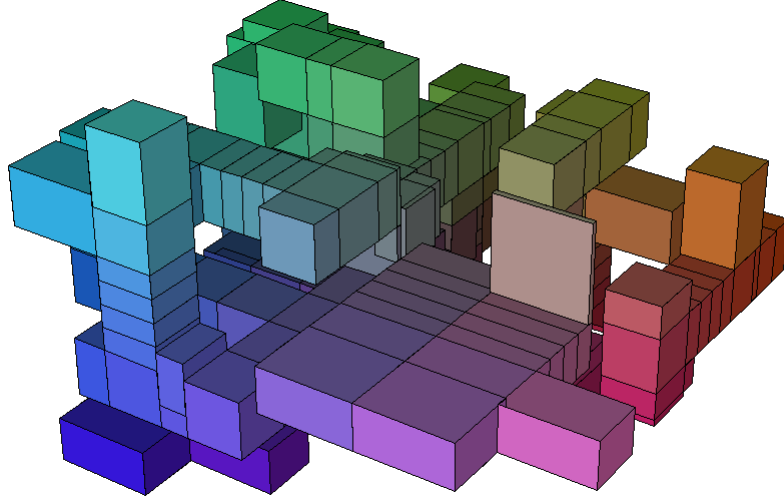
Figure 5.1: Sample design model. A colour gradient is overlaid to present depth.

to generate and evaluate. Another issue is that the evolved design models can generate embedded structures. This can potentially create non-visible geometry within the design needlessly. The most difficult challenge is the drastic increase in the size of the search space.

5.2   Design Languages

The dedication of the GP language for the creation of generative representations forces design operators to be separated into their own design languages, which the representations receive access to.

The developed evolutionary system in this research allows for the creation and alteration of various design languages. These design languages are composed of low-level operators, which interact with and alter the design models. Table 5.1 contains descriptions of these design operators. They are important to the system as they are what generate and alter the design models, guiding the models towards certain families of design styles.

The ability to easily create new design operators and languages within the system allows for more potential opportunities and designs. Only a small number of languages were created for this research and are outlined below, though the system is capable of many possibilities. The use of different design languages will result in the creation of differing sets and styles of artifacts.

Design languages are composed of a seed object and a set of design operators. The seed object is the starting geometry with which the system can begin to alter. The design operators are what are applied to the artifact to refactor its form.

Table 5.1: Design Operators

Operator	Description
Extrude	Extrudes the currently active face along its normal.
Left	Changes the currently active face to its neighbour on the left.
Right	Changes the currently active face to its neighbour on the right.
Up	Changes the currently active face to the neighbour above it.
Down	Changes the currently active face to the neighbour below it.
RotateCW	Rotates the currently active face clockwise.
RotateCCW	Rotates the currently active face counterclockwise.
Grow	Scales the currently active face larger.
Shrink	Scales the currently active face smaller.
Subdivide	Subdivides the current face into a set of smaller faces.

5.2.1   Growth Language

The growth language is the simplest language created for experimentation. It is based on a LOGO-style turtle drawing system, with extensions for rotation and scaling. This language is made up of the extrude, change-face, rotate and scale operators.

In this language, operators don't have parameters, they all have default argument values. Every call to the extrude operator is set to extrude one unit, rotates are set to ten degrees, and scales are set to ten percent of the current size.

5.2.2   P-Cuboid Language

The parametric-cuboid language allows access to parameter passing for the design operators. It is made up of the extrude and change-face operators. Parameters are generated using ephemeral random constants and are passed to those parameters. This language is able to grow the design by appending cuboid-like structures of various sizes to the current design.

5.2.3   P-Growth Language

This language extends the growth language, using the same operators set, but allowing for parameters to be passed to the design operators. It is similar to the growth language, but is unrestricted by parameters, allowing it to grow faster and in more unique manners.

5.3   Evaluation Functions

The system was designed and developed to allow for one-to-many fitness objectives, using the summed ranks technique for evaluation. Fitness functions can be selected and combined to guide the evolution in creating new types of design models.

Fitness functions were developed to lead in the creation of architectural form. Among these include form filling which guides the form to fill in towards a set of targets, and form repulsion which performs the opposite. Sun exposure, which guides form to expose itself towards or away from the influence of the sun. A small number of standard evaluation methods for models were implemented as well, including surface area and polygon count.

5.3.1 Form Filling

The form filling fitness function was designed to guide the generation of design model towards some rough form. Form filling works by inputting a set of three-dimensional *targets* into the system and having the generated geometry attempt to fill towards those targets.

The use of form filling is important, as it is likely the system would only generate very simple models - *single cubes for example* - to achieve the goals of other evaluation functions. It is meant to be paired with other evaluation functions, to generate some targeted form with aspects of the other functions.

The technique works by calculating distances between each polygon within the model and the targets. The algorithm matches each polygon to its closest target, adding the distance between the target and the polygon's furthest vertex to that targets score. As evolution progresses, each polygon is drawn in towards it closest target. This results in a set of faces which rougly approximate a set of targets. Pseudocode for the algorithm is presented in Figure 5.2.

---

1. Receive the list of Targets: T, for the current problem. Each member of T maintains three variables *totalDistance*, *amount* and *closestDistance*.

2. For each *face* in the design model:

   (a) For $i = 1$ to amount of T:

      i. Let *distances*$[i]$ be the distance between $T[i]$ and *face*. Distance is calculated as the distance between $T[i]$ and the furthest vertex of the *face*.

      ii. Set $T[i].closestDistance$ to the closest distance found so far for that target.

   (b) Let $x = i$ where, *distances*$[i]$ is the smallest value in *distances*.

   (c) Increment $T[x].amount$ by 1 for that target.

   (d) Increment $T[x].totalDistance$ by *distances*$[x]$.

3. For each t in T, perform the following:

   (a) if $t.amount = 0$: $t.totalDistance = t.closestDistance$ and $t.amount = 1$.

   (b) Increment *fitness* by $t.totalDistance/t.amount$.

4. Return the *fitness* value.

Figure 5.2: Pseudocode from the form filling fitness function.

5.3.2 Form Repulsion

Form repulsion is the opposite of the form filling function. It receives a set of three-dimensional *targets* and attempts to repel the form away. Form filling can be used to deter a form from either growing too large in some direction, or from filling in space between two form filling targets.

The technique works almost exactly the same to that of the form filling function. The main difference is in the way it calculates the distance between a target and a polygon. Instead of measuring the distance between the target and the furthest vertex, it measures the distance between the target and the closest point of the polygon. It then attempts to maximize those distances.

5.3.3 Sun Exposure

The Sun exposure fitness function attempts to examine an important factor in architectural problems, interaction with the Sun. It roughly calculates the amount of Sun exposure the design model is receiving. Figure 5.3 displays an example of this, where various faces receive different amounts of light intensity depending on their angular distance from the sun and whether they lie in shadow. Back-facing polygons in this example are not receiving any sunlight.
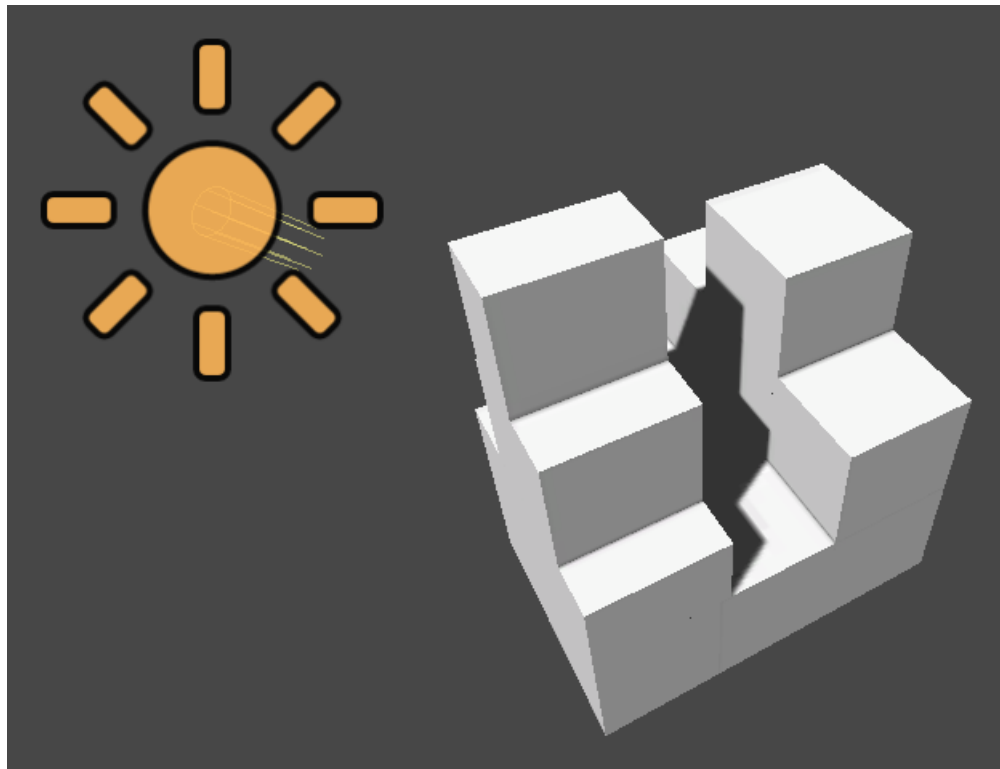
Figure 5.3: Design model affected by the Sun.

The sun exposure technique calculates a rough estimate of light intensity that the design model receives. Every polygon in the model is divided into a set of N by N points which are then examined. Taking this smaller set of samples helps to reduce execution times. Each point is checked to see if it is obscured by another model or itself, if not, its intensity score is taken into account. Figure 5.4 shares pseudocode associated with the sun exposure fitness function.

This objective can be calculated using one-to-many sun vectors, which allows the sun exposure to be averaged over a day. It can also be minimized or maximized, in the case of summer or winter. Finally, external objects can be added to cast shadows onto the design model.

---

1. Receive a vector: Sun, for the current problem.

2. Let N be the amount of subdivisions per face.

3. For each $face$ in the design model:

    (a) Let $surfaceIntensity = 0$.

    (b) Determine the $angle$ between $face.normal$ and Sun.

    (c) If $angle \leq 0$: Continue to the next $face$.

    (d) Let $intensity = angle * face.SurfaceArea/N^2$.

    (e) Create a set $subDivision$ of N by N points, distributed evenly over the $face$.

    (f) For each SD in $subDivision$:

        i. Create a Ray starting at SD in the direction of Sun.

        ii. Cast the Ray and determine if it intersects any other face in the design model.

        iii. If the ray is not blocked: $surfaceIntensity+ = intensity$.

    (g) $fitness+ = surfaceIntensity$

4. Return the $fitness$ value.

Figure 5.4: Pseudocode from the sun exposure fitness function.

5.3.4   Surface Area

This function returns the total surface area of the design model being examined. It can either attempt to be minimized or maximized, which results in smaller and larger polygons.

5.3.5   Polygon Count

The polygon count fitness function is quite simple in that it returns the number of polygons in the model. This is an effective strategy for attempting to minimize the size and geometrical complexity of the generated artifacts, allowing for shorter execution times.

Chapter 6

Experiments with Generative Representations

This chapter contains an overview of the first set of experiments carried out. These experiments are meant to validate that the developed system is able to encode generative representations and ensure that it is able to generate design structures. The first problem examined was developed to be a simple design problem, that each representation could potentially do well in.

All experiments in this research were executed on a computing cluster in the computer science department at Brock University. The cluster is composed of three sets of resources. The first set of resources are AMD Phemom II 1090T (3.2GHz Six-core) with 8GB of DDR3 RAM. The second set are Intel Core i7-920 (2.66GHz Quad-core) with 12GB of DDR3 RAM. The final set is made up of Intel Core i5-2500 (3.3GHz Quad-core) with 4GB of RAM.

6.1   Experimental Settings

The goal of this set of experiments is to investigate the use of generative representations in a new problem area, with focus on examining their performance in comparison to one another, and inspecting the models they generate from a subjective, aesthetic perspective. The five representations that are to be examined are Simple (S), Modular (M), Modular-Reuse (MR), Modular-Hierarchical (MH) and Modular-Reuse-Hierarchical (MRH). These are based on Hornby's representations used on a voxel-based table design problem [17].

Experimentation will be performed on a simple space-filling problem outlined below. This will determine if any of the representations clearly outperforms the others on a simple problem. It will also determine if the space-filling evaluation strategy is valid for generating forms that resembles the target point set, as well as if the system is able to generate diversity in its solutions. It will be done in

both single and multi-objective settings, to further explore diversity in design solutions and to derive any early comparisons on performance in multi-objective problems. Table 6.1 presents a summary of baseline parameters used for all experiments in this research. Many of these parameters were determined to be effective in preliminary experiments. The simplest design language was selected as a means to attempt to level the playing field amongst the representations.

Table 6.1: Baseline GP And Encoding Parameters.

Parameter	Value	GR Parameter	Value
Generations	3000	Procedure Amount	7
Population Size	100	Procedure Arity	2
Crossover Rate	90%	Condition Pairs	2
Tournament Size	4	Design Language	Growth
Elite Count	1	Seed Object	Cube
Maximum Crossover Depth	10		(1x1x1 units)
Mutation Rate	10%		
Maximum Mutation Depth	17		
Prob. of Terminals in Cross/Mut	10%		
Initialization Method	Half-and-half		
Tree Grow Max / Min	5 / 5		
Tree Full Max / Min	12 / 5		
Experimental Trials	30		

6.2 The Arch Problem

The arch problem is designed as a simple yet non-trivial evolutionary design problem. The goal is to have an evolutionary system generate an arch whose geometry approximates a set of target points. Figure 6.1 displays a set of nine target points that the system will receive, as shown from an orthogonal view. These points all lie within the same plane, along the $z = 0$ axis, making it a simplified version of the space filling problem. It is a minimization problem, where the system must attempt to reduce the error between the evolved shape and the target point set.

Three sizes of the arch problem are considered. These sizes are shown in Table 6.2 to give the reader an idea of approximate error values from generated models. This will allow for comparison on the scalability of the generative representations if needed.

Table 6.2: Relative sizes of target arches.

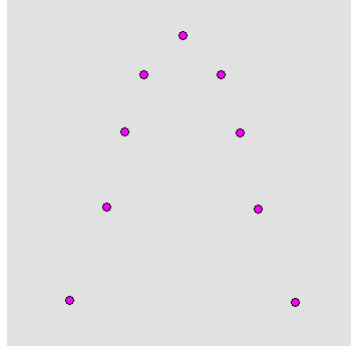| Arch Label	Height	Width
Small	28 units	24 units
Medium	56 units	48 units
Large	112 units	96 units

Figure 6.1: Arch target points for form filling evaluation, as seen from orthogonal view.

6.3 Single-Objective Results

Experiments were carried out on the small, medium and large arch problems, using only the single form-filling fitness evaluation. They were performed for each of the five representations, totalling fifteen experiments.

A graph detailing the evolutionary progress of the large arch experiment can be seen in Figure 6.2, spanning 3000 generations. Here we can clearly see that the simple representation is outperformed by the four generative representations. Amongst the four generative representations, there does not seem to be a clear winner. MRH is able to discover good solutions very quickly, achieving the top place in best scores but fourth in population average. M, MR & MH all perform similarly in their evolutionary progress in this problem. Performance graphs of the small (Figure A.1) and medium (Figure A.2) arch experiments are shown in the appendix.

Table 6.3 displays the average of mean and average of best scores of the final population from the fifteen experiments, taken over thirty trials. Here we see that all five representations perform relatively close in the small arch problem. In the medium arch problem, the simple representation begins to diverge from the four generative representations. In the large arch problem, the simple representation garners approximately double the score of the other four representations. MH performs the best overall in all categories except one, where MRH outperformed it in best scores on the large arch problem.

Table 6.4 presents the p-value scores, calculated using a one-tailed t-test on the average of best scores in the large arch problem. A p-value below 0.05 indicates that the difference in fitness values is statistically significant. The large arch problem was selected for this diagram as it was the most difficult. In this case, we see that all representations outperform S and that MRH outperforms all others. An interesting point to note, is how the MRH representation tends to have high average scores in comparison to the other generative representations.

Images of the best design objects generated by each representation for the small arch problem
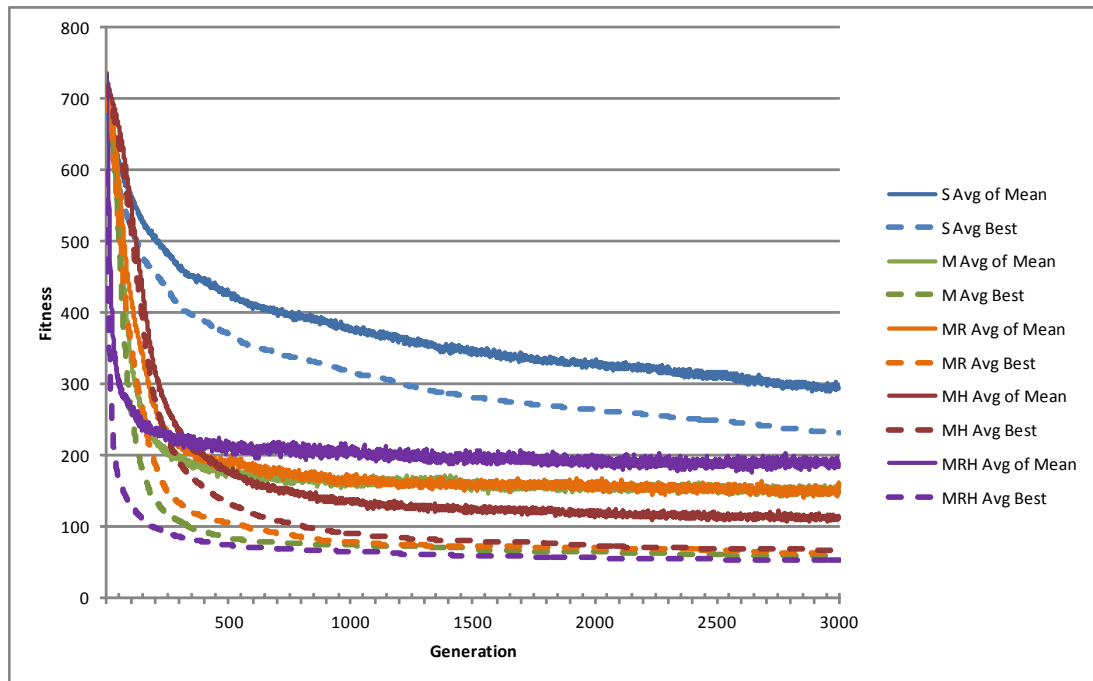
Figure 6.2: Performance graph from the large arch experiment. Displaying the average of mean and best fitness over 3000 generations from 30 trials.

are shown in Figure 6.3. The design structures created do indeed resemble arches, verifying that the form-filling function does as intended.

Table 6.3: Average and best fitness from the five representations on the small, medium & large arch problem over 30 trials. (Best scores in boldface.)

Rep	Small		Medium		Large	
	Average	Best	Average	Best	Average	Best
Simple	45.0	18.8	111.5	72.5	292.6	231.3
M	30.2	9.2	60.4	21.4	150.6	58.3
MR	28.8	12.1	72.5	31.5	145.4	61.8
MH	**23.8**	**8.5**	**53.4**	**19.1**	**112.8**	66.5
MRH	47.0	11.4	91.2	26.0	188.4	**52.1**

Table 6.4: P-value scores from one-tailed t-test, comparing encoding's average of best scores on the large arch problem. A p-value less than 0.05 indicates statistical significance that A differs from B.

		B				
		S	**M**	**MR**	**MH**	**MHR**
	S		-	-	-	-
	M	0.0			-	-
A	**MR**	0.0	-		-	-
	MH	0.0	-	-		-
	MRH	0.0	0.041	0.009	0.018	

6.4 Multi-Objective Results

These experiments are intended to observe how the various representations would perform in multi-objective problems, as well as how added objectives might alter the appearance of generated design objects. The experiments were all performed using the medium arch as a target.

This set of experiments attempts to minimize arch error and reduce polygon count. Experimentation was performed on the evolutionary system using the form-filling function and polygon count as two objectives. Five experiments took place, one for each representation, 30 trials for each experiment.

The evolutionary performance of these experiments are graphed in Figure 6.4. Average arch scores and average polygon counts are shown over the course of 3000 generations. The simple representation performs the poorest in arch score but best in polygon score. The MH representation does the opposite, performing best in arch score but worst in polygon count. MRH seems to perform the worst overall, placing fourth in both objectives.

A scatter diagram is shown in Figure 6.5, plotting the best solutions of the final population
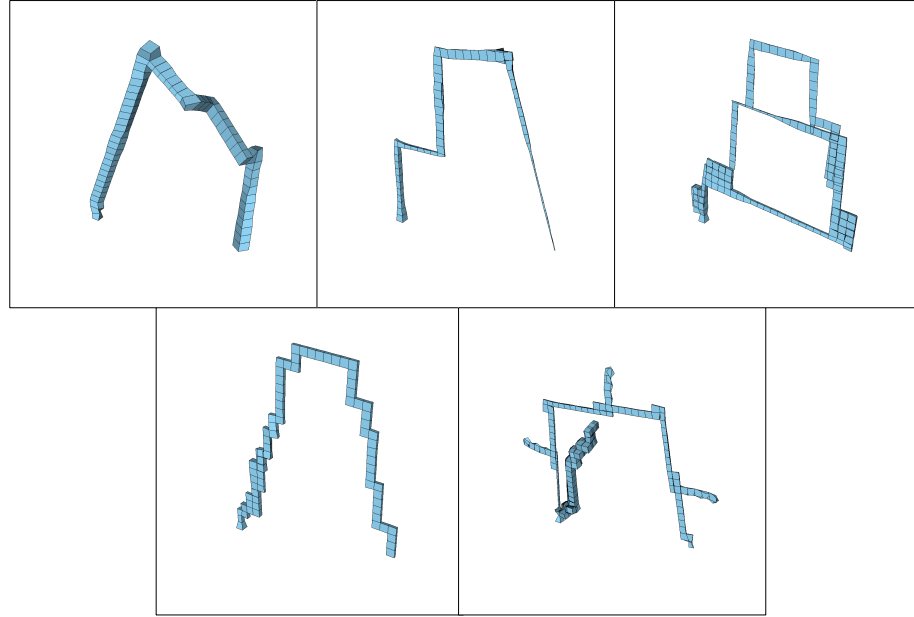
Figure 6.3: Images of best results generated from the S, M, MR (top: left to right), MH and MRH representations (bottom: left to right) in the small arch problem.
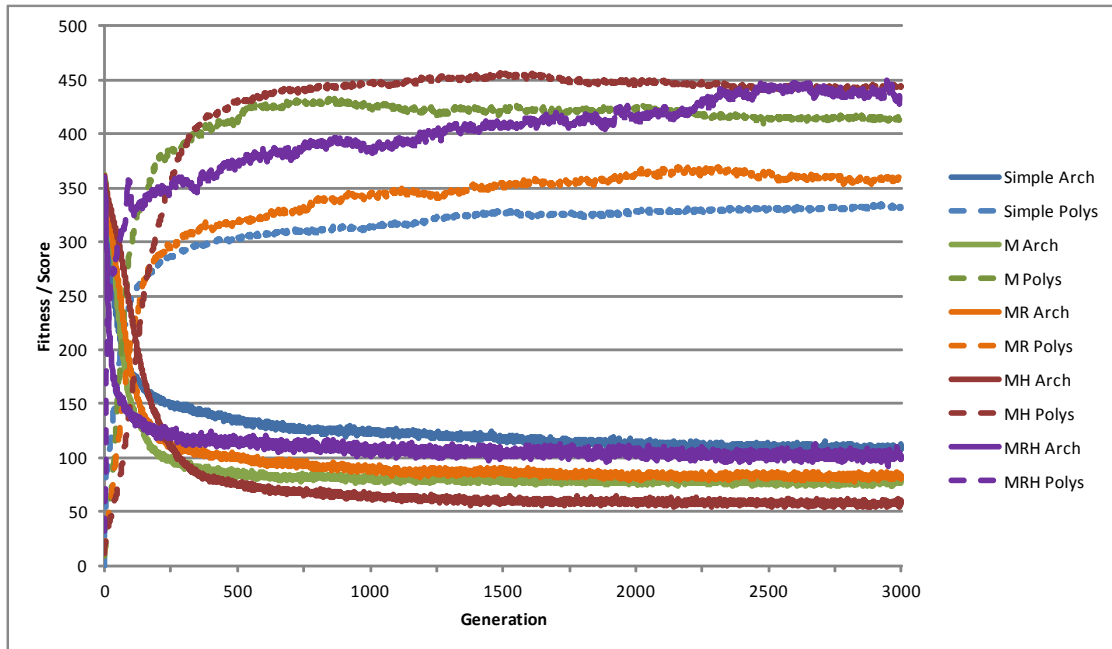
Figure 6.4: Performange graph of arch error and polygon count on the medium arch problem. Displaying the average of mean over 3000 generations from 30 trials.

found and how they scored in arch error versus polygon count. Of the 750 solutions shown, each of the five representations contributed 150, five best solutions over 30 trials for each. Similar to the results of the single-objective experiments, we can see that the simple representation is outperformed by the four generative representations when best solutions are concerned. Of the four generative representations it appears that the MRH solutions are clustered in the optimal locations along the pareto front.

Figure 6.6 contains examples of some of the best evolved models. MRH appears to create design objects which are not arches, favoring polygon count in this case.
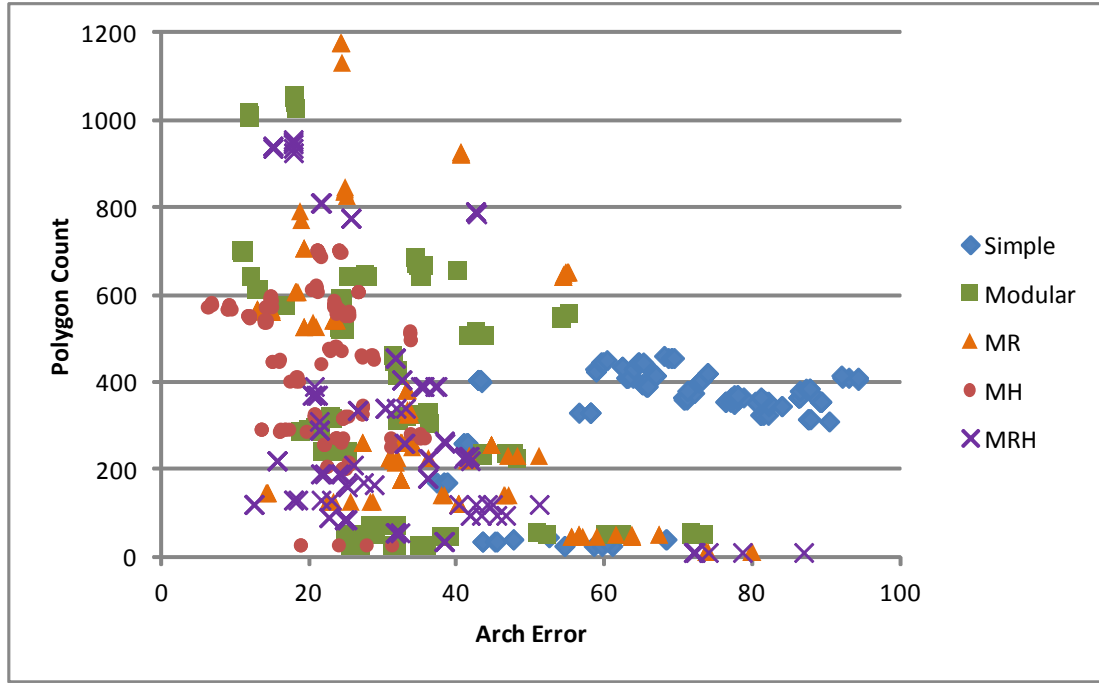
Figure 6.5: Distribution of 750 total solutions, showing 150 of each representation in the arch error versus polygon count experiment.

6.5 Conclusions

Following these experiments, one could conclude that the generative representations do outperform the simple representation. Though it is not quite clear from these experiments which of the generative representations is best suited for this problem domain. This calls for further exploration in a more difficult problem set.
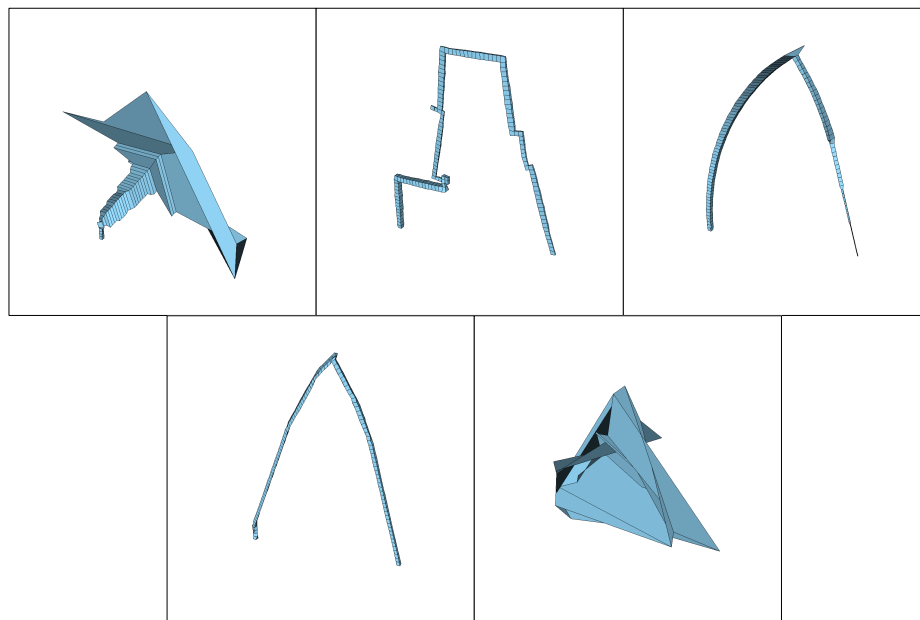
Figure 6.6: Images of best results generated from the S, M, MR (top: left to right), MH and MRH representations (top: left to right), measuring arch error and polygon count on the medium arch problem.

Equipped with a very simple design language, the evolutionary system was successful in automatically generating form that resembled the target point set. The designs that were created using the system were generated as single, water-tight objects. This fact is shown in Figure 6.7, which shows two evolved arches that were rendered using a three-dimensional printer. Generated arches came in a wide variety of styles, as is shown in Figure 6.8, validating its usefulness in design problems as an inspiration tool. The four generative representations did reasonably well in discovering an arch for all sizes of the arch problem, the simple representation was not able to do as well in larger versions of the problem. By adding more objectives to the form-filling problem, we were able to generate more specific designs, showing great promise in what is possible with this technology.

Figure 6.7: Generated arches and their three-dimensional print outs. Printing resin remains on the right print for demonstration purposes.
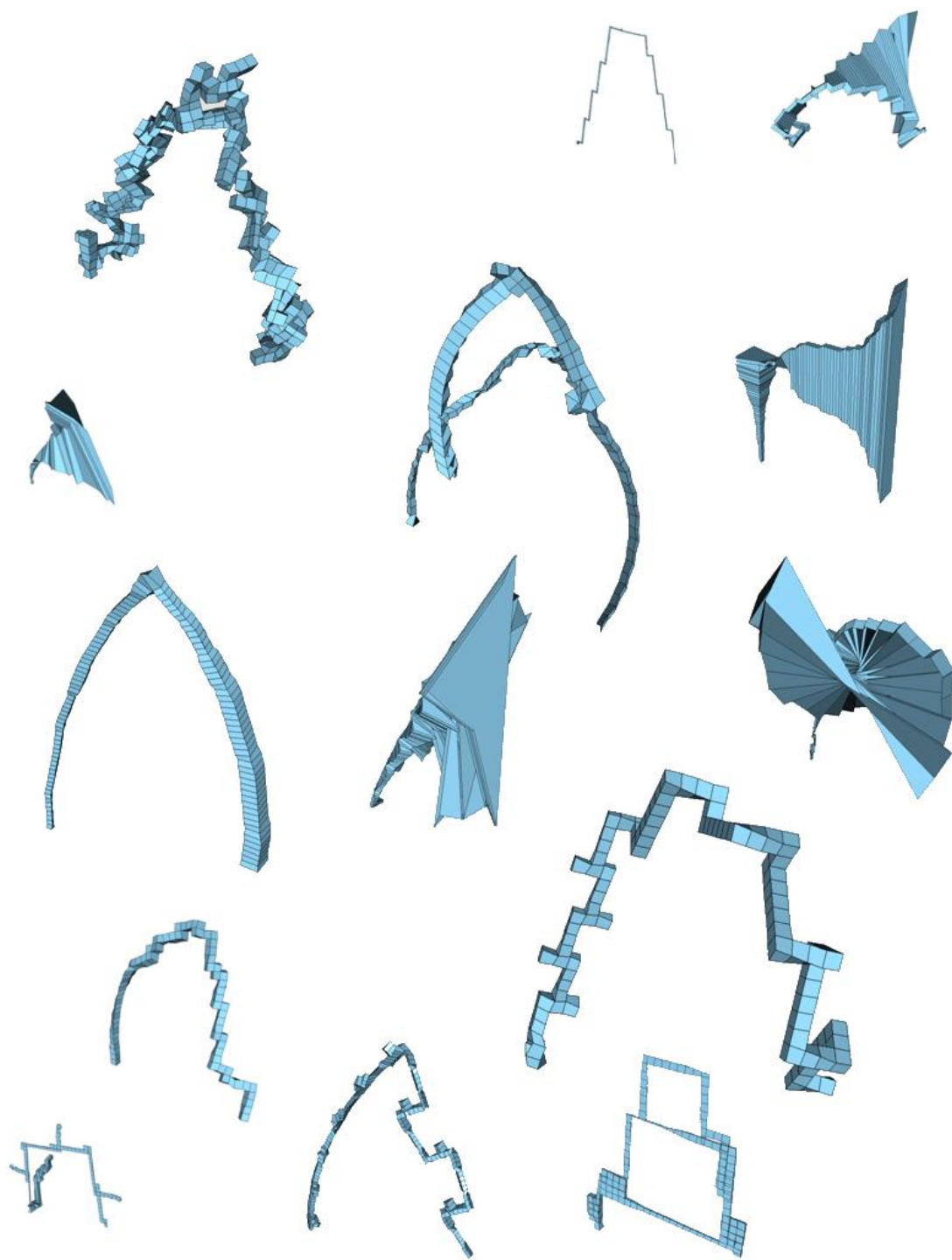
Figure 6.8: Variety of arch designs.

Chapter 7

Advanced Experiments with Generative Representations

This chapter contains details of the experimental setup of a more advanced evoluationary design problem and presents results using generative representations. After validating the developed system and providing proof f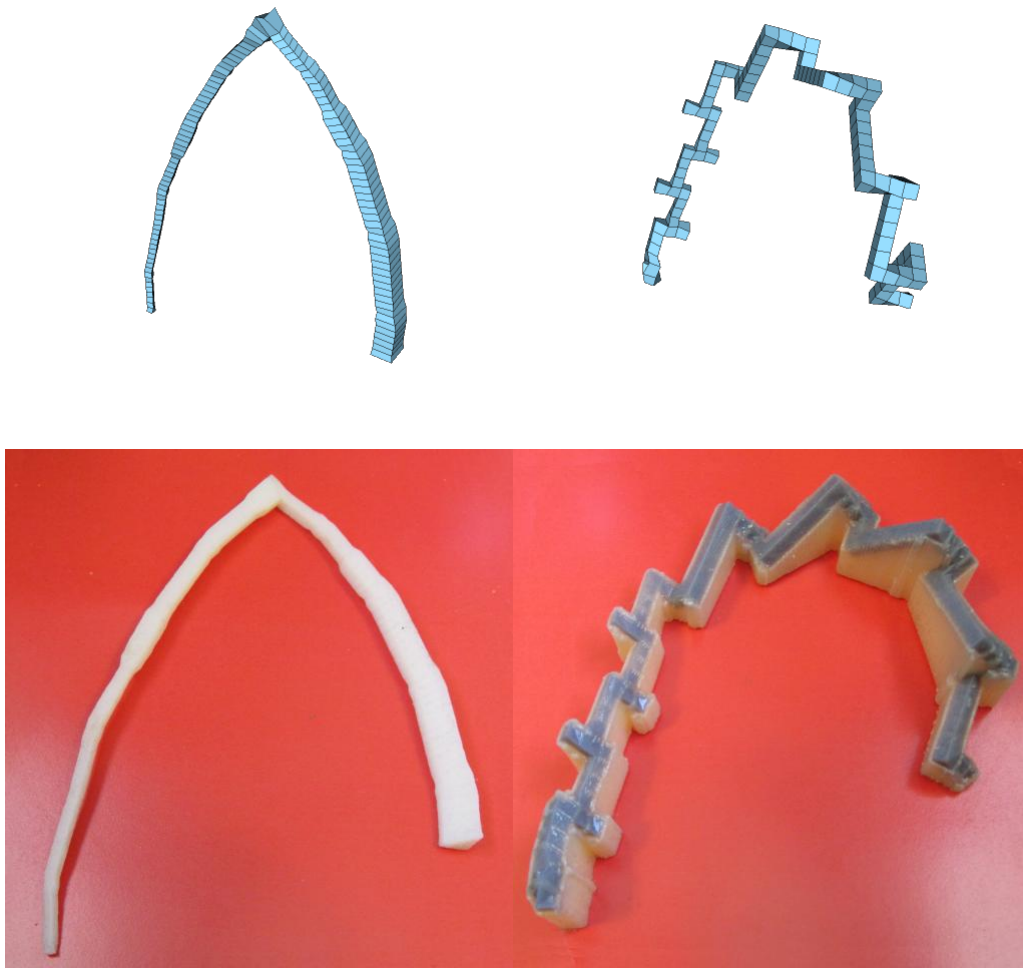or the need of generative representations in the previous chapter, a set of more difficult space-filling problems was developed. These problems will create a larger disparity in the results from the generative representations, allowing for the examination of complexity metrics.

7.1 Experimental Settings

The goal of these experiments are to facilitate a more competitive environment for the generative representations. We would like to be able to discern which features of complexity are important to the field of evolutionary design, specifically for three-dimensional form generation. We also wish to utilize the parametric languages, allowing the GP system access to more advanced operations. Rather than provide a target shape like the arch problem, we would like to observe the style of form to be generated from random targets. Details for the random target problem are provided below.

Table 7.1 contains the GP and encoding parameters used in these experiments. The number of generations was changed to 500, as it was seen in the previous chapter that 3000 was too many. The design language is set to the parametric growth language, allowing design operators to execute with arguments.

Table 7.1: GP And Encoding Parameters for advanced form filling experiments.

Parameter	Value	GR Parameter	Value
Generations	500	Procedure Amount	7
Population Size	100	Procedure Arity	2
Crossover Rate	90%	Condition Pairs	2
Tournament Size	4	Design Language	Parametric Growth
Elite Count	1	Symbol Arity	1
Experimental Trials	30	Seed Object	Cube
			(1x1x1 units)

7.2   Random Targets Problem

The random targets problem is designed as a difficult evolutionary design problem. The goal of this problem is to challenge the evolutionary system to generate form, whose geometry approximates a set of N random points. Fifty points were generated at random prior to experimentation and are the same for all trials, they are shown in the appendix in Table A.1. The points are distributed in three-dimensions, ranging from -50 to +50 for x and z and 0 to +50 for y. The form filling fitness function is used for this problem and is presented in Figure 5.2. Four sets of experiments are undertaken using five, ten, twenty and fifty target points. Figure 7.1 displays the targets for the ten random point problem.

Random points are used instead of a preconceived construct to gain a better understanding of the style of design objects that are created using these techniques. Another reason for this is to eliminate any potential bias that could be present by selecting an object that would be better suited for a particular representation.
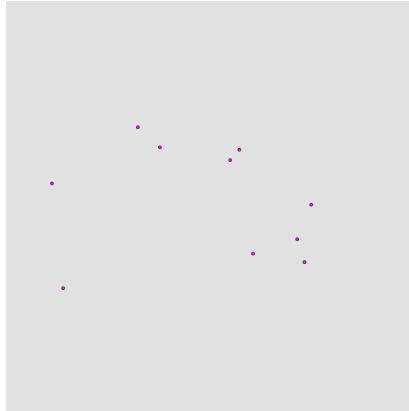

Figure 7.1: Ten random target points as seen from the perspective view.

7.3 Results

7.3.1 Performance

Experiments were carried out on the five, ten, twenty and fifty random point experiments. A graph of the evolutionary progress is shown in Figure 7.2 for the fifty point problem. Graphs for the five, ten & twenty experiments can be found in the appendix (Figure A.5 - A.7).

It can be seen from the graph that MRH is once again very quick to gain ground in fitness score, finishing with the best results in both average of mean and average of best. The simple representation performs quite poorly, never dropping below a fitness score of 1000, which is double that of any other representation. MR & MH appear to be neck and neck throughout the evolutionary process, with the Modular representation outperforming both by a small amount.
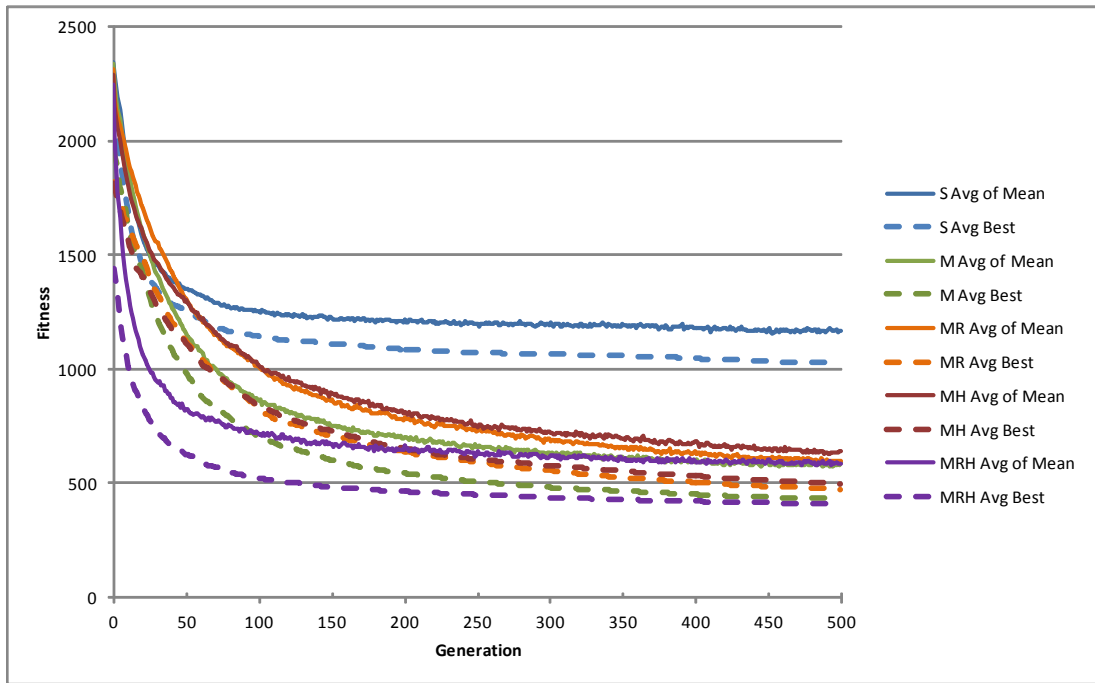
Figure 7.2: Performance graph from the fifty random point experiment. Displaying the average of mean and best fitness over 500 generations from 30 trials.

Table 7.2 contains the final population results from the four experiments. It can be seen that the simple representation performs the worst in each field and is not able to scale effectively to larger problems. MH performs best overall in the five and ten point experiments, with the M and MRH representations performing very closely. In the twenty point experiment, MRH takes over as the leader, with both M and MH doing nearly as well. In the fifty point experiment MRH clearly

outperforms all others, showing its ability to scale well to larger and more complex problems. The MRH representation once again maintains a larger difference in average and best scores in comparison to the other representations.

A one-tailed t-test was performed on the results from the fifty point experiment and a p-value is shown in Table 7.3. A value of less than 0.05 indicates that A is significantly different from B.

Table 7.2: Average and best fitness from the five representations on the five, ten, twenty and fifty random point problem over 30 trials. (Best scores in boldface.)

Rep	Five Point		Ten Point		Twenty Point		Fifty Point	
	Average	Best	Average	Best	Average	Best	Average	Best
Simple	161.1	147.6	267.2	236.2	483.4	429.8	1165.9	1026.3
M	103.7	77.2	169.7	130.9	290.3	215.8	592.4	431.4
MR	112.8	92.9	186.3	150.3	299.7	242.2	595.5	473.5
MH	**94.4**	**70.8**	**164.2**	**128.4**	**279.5**	216.7	640.3	497.9
MRH	109.5	73.0	182.5	133.4	292.1	**210.9**	**586.4**	**409.4**

Table 7.3: P-value scores from one-tailed t-test, comparing significance of encoding's average of best scores on the fifty random point problem. A p-value less than 0.05 indicates statistical significance that A differs from B.

		B				
		S	**M**	**MR**	**MH**	**MHR**
	S		-	-	-	-
	M	0.0			-	-
A	**MR**	0.0	0.0		-	-
	MH	0.0	0.0	0.024		-
	MRH	0.0	0.036	0.0	0.0	

7.3.2 Design Objects

Samples of the best individuals are shown in Figure 7.3 through Figure 7.6 for each of the four experiments. From these images we can see that the simple representation is not able to handle this problem, generating stick-like design objects rather than voluminous ones. While MH appears to achieve high fitness scores, it seems to do so by creating large container structures to envelope all target points, losing and geometrical detail. The M and MR representations are able to generate interesting results, though they do not contain the geometric fidelity of the MRH representation. The design objects created by the MRH encoding are very compelling, creating sub-structures and detailed geometry that are appealing to the eye.

Figure 7.3: Images of best results generated from the S, M, MR, MH and MRH representations (ordered left to right) in the *Five* random point problem.

Figure 7.4: Images of best results generated from the S, M, MR, MH and MRH representations (ordered left to right) in the *Ten* random point problem.

Figure 7.5: Images of best results generated from the S, M, MR, MH and MRH representations (ordered left to right) in the *Twenty* random point problem.
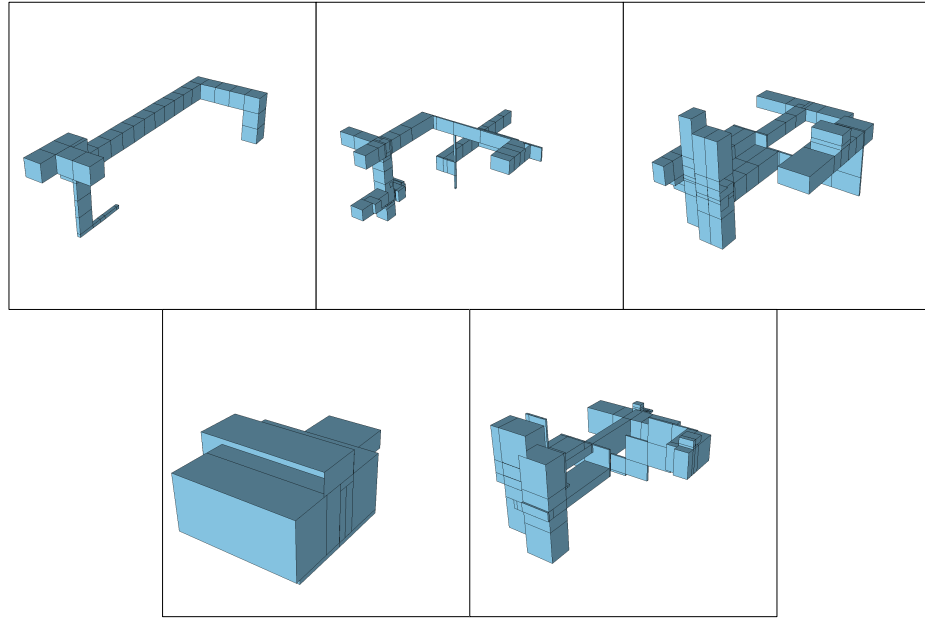

Figure 7.6: Images of best results generated from the S, M, MR, MH and MRH representations (ordered left to right) in the *Fifty* random point problem.
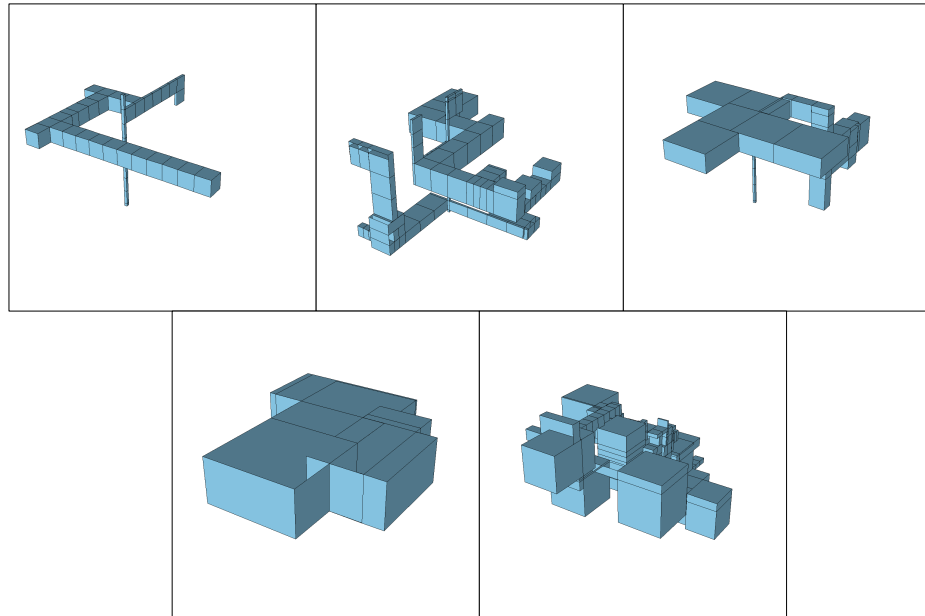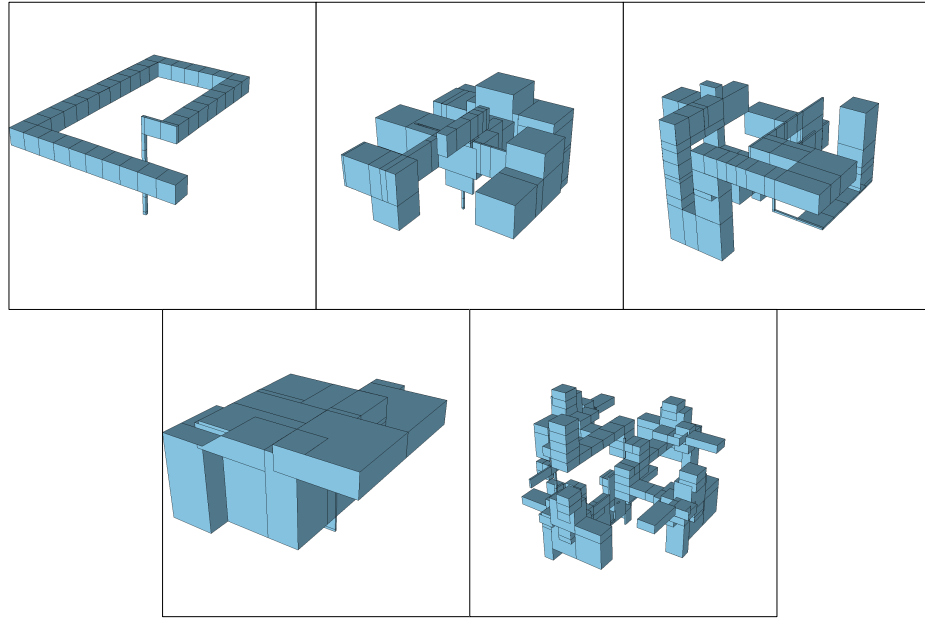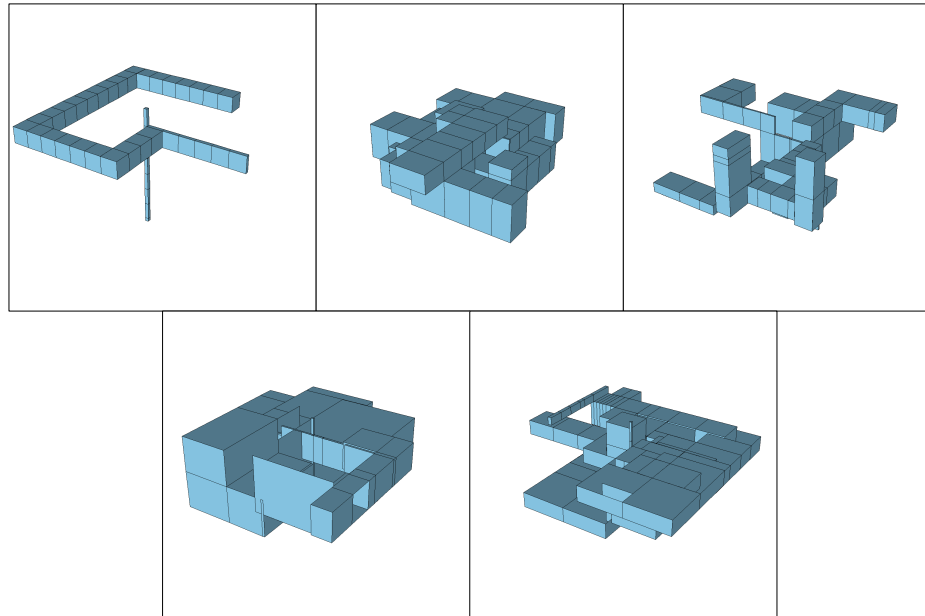
7.3.3   Complexity

Measures of complexity were calculated for all experiments conducted in this chapter. Figure 7.7 displays measures of genotype and phenotype length, modularity of both the program and design encoding, reuse, hierarchy, AIC and sophistication from the fifty point experiment. Each scatter diagram contains nodes from each individual of the final population from the 30 MRH trials, totalling 3000 individuals.

The lengths of the genotypes typically range from approximately 500 to 4000. After transformation into a design string, they typically range from 25000 to 150000 operators in length. The ability to create large design encodings from small genotypes is one of the major strengths of the generative representations. Lengths do not seem to play a direct role in determining fitness, though it seems that for larger values you are less likely to get mid to high fitness scores.

Modularity scores are calculated for both the representation and the design encoding. Values of M_p between five and seven appear to perform well and any individual with a score less than five seems to perform poorly. For the M_d metric, scores range up to 4000, with well performing individuals at all values. There appears to be a cluster of strong performing individuals with a high M_d score.

Measures of *reuse* range from 0 to 50, with a pocket of very strong individuals sitting at around a reuse of 40 similar to that of the M_d metric. *Hierarchy* of all strong individuals is in the range of 5 to 7, appearing to be very similar to the M_p measure.

AIC scores range between 150 and 250, and appears to also have a grouping of strong performing individuals where the score reaches 250. Sophistication does not appear to have any clear pattern in this experiment.

It is important to note that these measurements are taken on the final population of the MRH experiment. Typically, the final population will contain a set of relatively good solutions. This fact makes it more difficult to draw conclusions based on the measurements. It appears in these experiments that the measures of *AIC*, *reuse* and M_d were the most useful. They clearly show that the best individuals in the population were enabled by these features of complexity. Genotype and Phenotype length appear similar to the previous three, but it is more subtle. *Sophistication* did poorly as a measurement of complexity in this case, showing no clear pattern. M_p and *Hierarchy* display interesting results. In these cases, seven was the highest score that could be achieved for this experiment, and the populations tended towards higher values of M_p and *Hierarchy*. The measurements explored here, help to verify that modularity, reuse, hierarchy and AIC are important features in encoding individuals for evolutionary design.
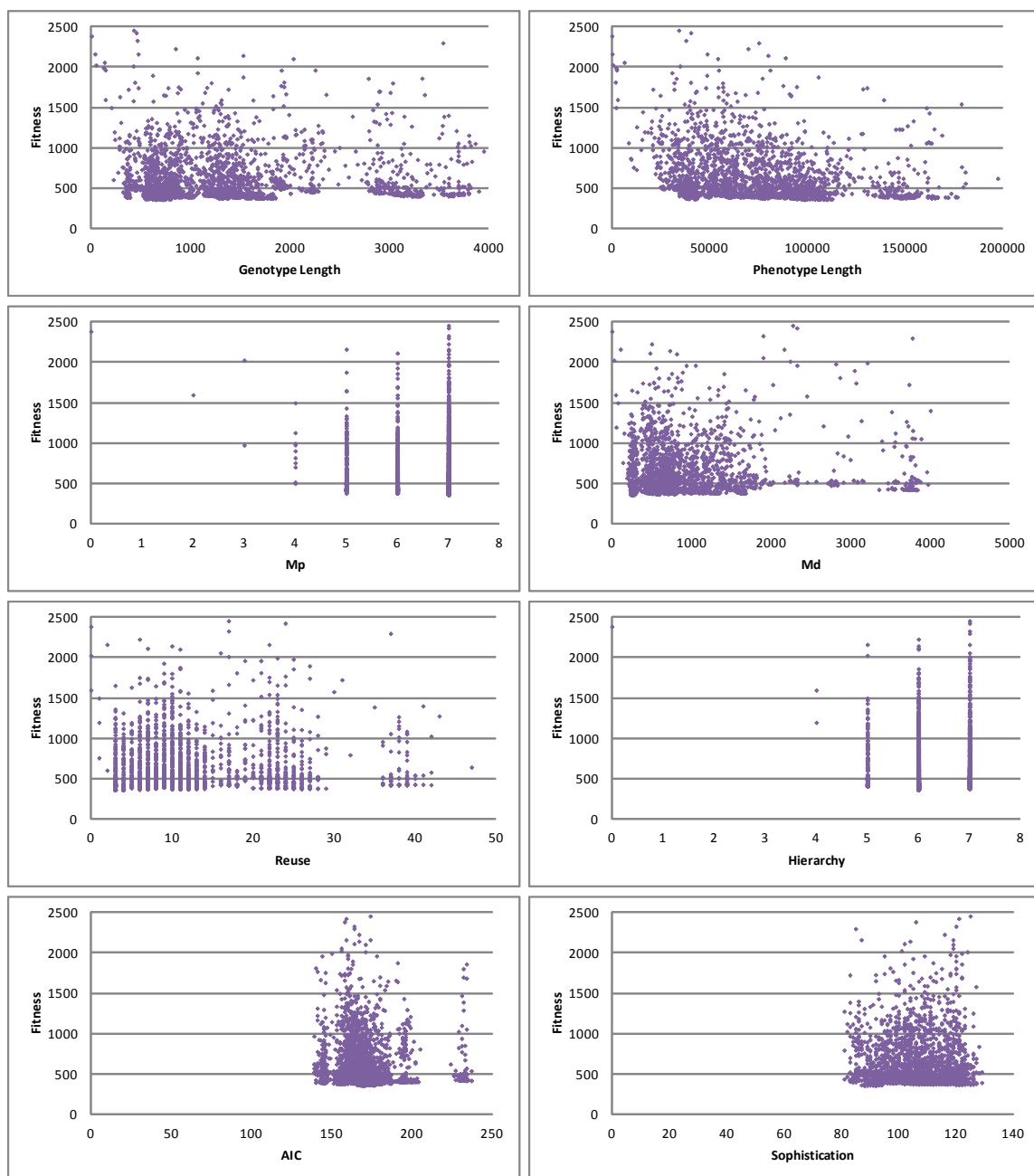
Figure 7.7: Measured values of genotype length, phenotype length, modularity, reuse, hierarchy, AIC and sophistication for MRH, from the fifty point experiment.

7.4 Conclusions

The system was capable of scaling to the complexity of a more difficult space filling problem. Further proof was provided that the Simple representation is not capable of scaling to more difficult and complex problems. M, MR, MH were able to achieve good fitness scores in the five and ten point experiments, but their design objects were unprovocative. MRH was able to perform well in all of the experiments performed in this chapter and performed best in the most difficult problems. Not only did it perform well, it also generated the most captivating geometry amongst the representations.

While there is no clear-cut pattern from the charts on complexity, one assume that as the population evolves it will tend to grow in the areas of M_p and *hierarchy*, cutting individuals with lower values in these metrics from the final population. Also that while it is possible to achieve strong fitness scores and low measures of M_d and *reuse*, achieving high values in these measures will result in overall stronger individuals.

Appealing and novel architectural structures begin to appear when using the MRH encoding. Images from various perspectives for the best and second best individuals from the ten point experiment are shown in Figure 7.8 & Figure 7.9. The two best results are very similar in fitness score, but differ vastly in form. This is further proof of the system's ability to generate diverse and novel design objects.
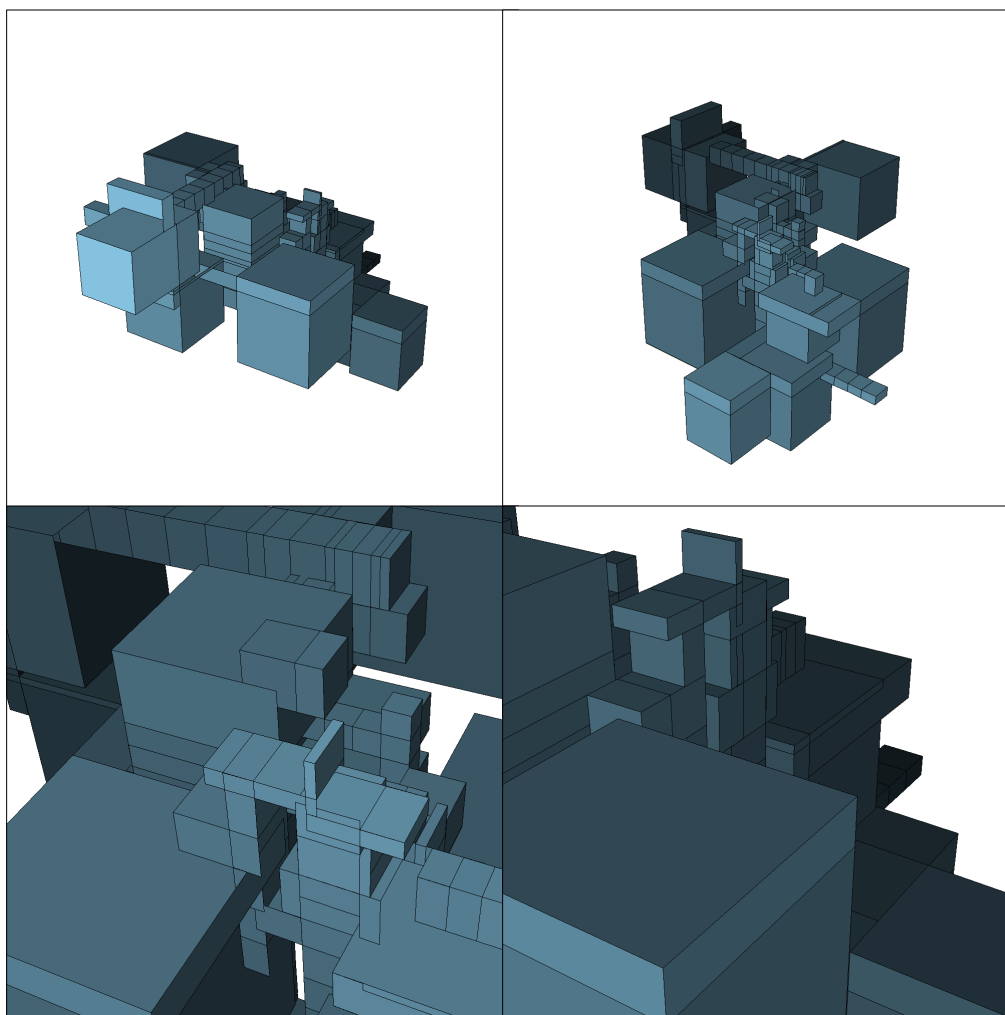
Figure 7.8: Perspective views and close-ups of the best MRH individual from the ten point experiment.
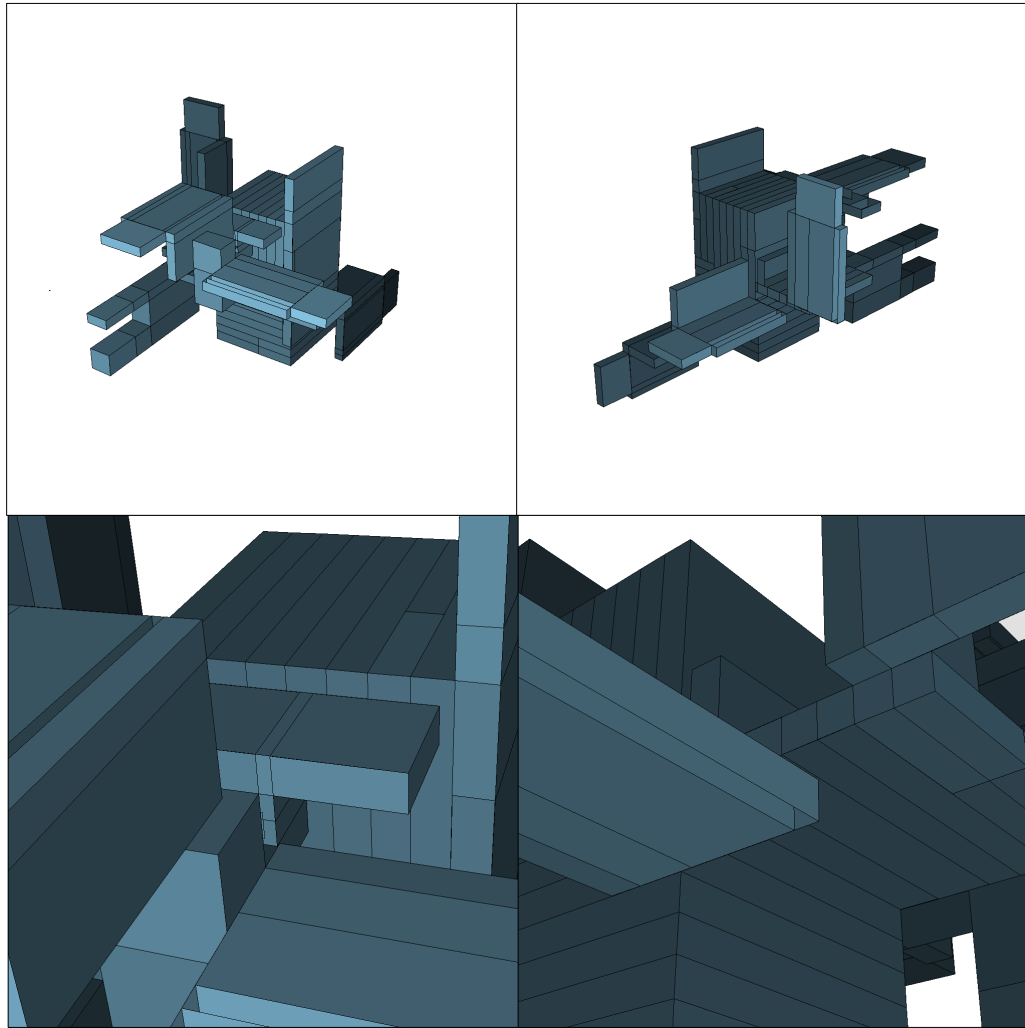
Figure 7.9: Perspective views and close-ups of the second best MRH individual from the ten point experiment.

Chapter 8

Generated Architecture with External Factors

This chapter begins to explore architectural problems in more detail. From previous chapters, we have shown that the MRH representation performs well in this problem domain, and so we will only be using that representation moving forward. Here, we will examine the generation of form with influence from the Sun as an external factor. Experimental setup, problem descriptions and results are presented below.

8.1 Experimental Settings

These experiments are designed to examine how external factors affect the generated design structures. We will attempt to examine architectural problems, applying the developed system for the automatic generation of three-dimensional models that are optimized for Sun exposure. In this chapter, only the MRH representation will be examined.

Table 8.1 contains the parameters used for these experiments. As these experiments are computationally intensive to execute, the experimental trials were reduced to 15 and generations reduced to 100 for all experiments except the single sun experiment.

8.2 Sun Exposure Problems

A set of problems which examine the Sun in various ways were designed. The generated designs are influenced by multiple objectives, these being the Sun exposure and ten random target fitness functions. These two fitness functions are shown in more detail in Figures 5.2 and 5.4. A series of Sun vectors were created to simulate the change of location in the sky over the course of a day, as

Table 8.1: GP And Encoding Parameters for Sun exposure experiments.

Parameter	Value	GR Parameter	Value
Generations	500, 100	**Procedure Amount**	7
Population Size	100	**Procedure Arity**	2
Crossover Rate	90%	**Condition Pairs**	2
Tournament Size	4	**Design Language**	P-Cuboid
Elite Count	1	**Symbol Arity**	1
Experimental Trials	15+	**Seed Object**	Cube
			(1x1x1 units)

well as height of the Sun in the summer and winter seasons. These Sun vectors are shown in the appendix in Table A.2. In passive solar building design, the architectural form is made to collect solar energy in the form of heat during the winter and reject it during the summer.

The first experiment eases into this set of problems, only examining one Sun vector. Comparisons will be made against the single objective ten random targets experiment, to discern how the Sun exposure fitness changes the look of the design models and how the Sun exposure objective draws from the form filling objective.

The second experiment expands by taking into account multiple positions of the Sun in the sky. We will attempt to maximize the sun exposure of the designs here, simulating Winter time. Figure 8.1 shows how multiple Sun positions are setup, to interact with a design structure.
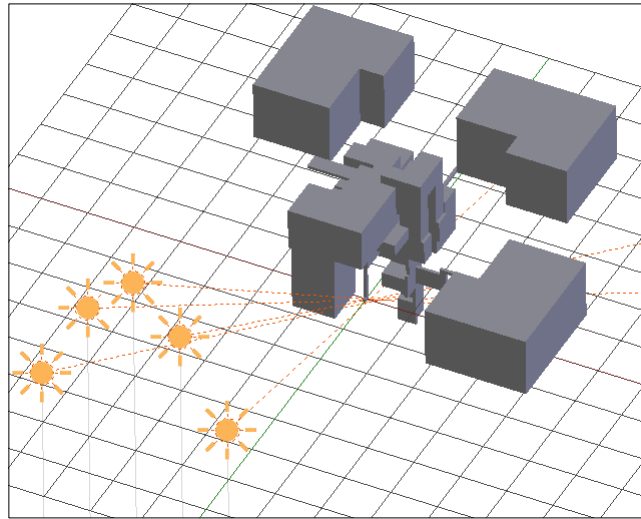

Figure 8.1: Multiple Sun position experimental setup.

The final experiment adds three extra structures externally, blocking sunlight to the design models. We hope to see that the generated architectures grow around the blocking models, to

maximize exposure to sunlight.

These three experiments are summarized in Table 8.2, describing all objectives for each experiment.

Table 8.2: Summary of Sun exposure experiments and their evaluation objectives.

Experiment	Description
Single Sun	Evaluated on two objectives, the ten point target form filling objective, and Sun exposure based on a snapshot in time at noon.
Winter Sun	Extends the single Sun experiment, taking into account five positions of the Sun and averaging their intensity scores.
Shadow City	Further extends the winter Sun experiment, adding three solar obstructing objects to the multiple Sun exposure objective, and an objective to minimize surface area.

8.3   Single Sun Results

Experiments were performed using the Sun exposure fitness, with the goal of maximizing sun exposure towards the noon Sun. In Figure 8.2 we can see the evoluationary progress of the single Sun experiment, with reference to the same experiment without the Sun exposure objective for comparison. In this graph we can see that the system scales to the Sun exposure fitness well, as the difference in fitting scores between the single and multi-objective runs are very small throughout the evolutionary run. Table 8.3 displays the final population scores for both the single and multi-objective runs. We see that the differences in final scores are minimal, with average scores of 180 and best scores of 130.

Table 8.3: Average and best fitness from the single sun experiment over 30 trials.

Type	Form Fitting		Sun Exposure
	Average	Best	Average
SO	182.5	133.4	-
MO	183.5	135.9	6056.0

Figure 8.3 displays a scatter diagram of the best ten individuals from each trial. Four example design models are presented, showing what the various clusters of population results look like. The two design models above the diagram achieved high Sun exposure scores, and the two below achieved low Sun exposure scores. We can see that the design architectures that achieved high Sun scores are more volumous, that they show more surface area towards the Sun, and that they have staircase-like features.
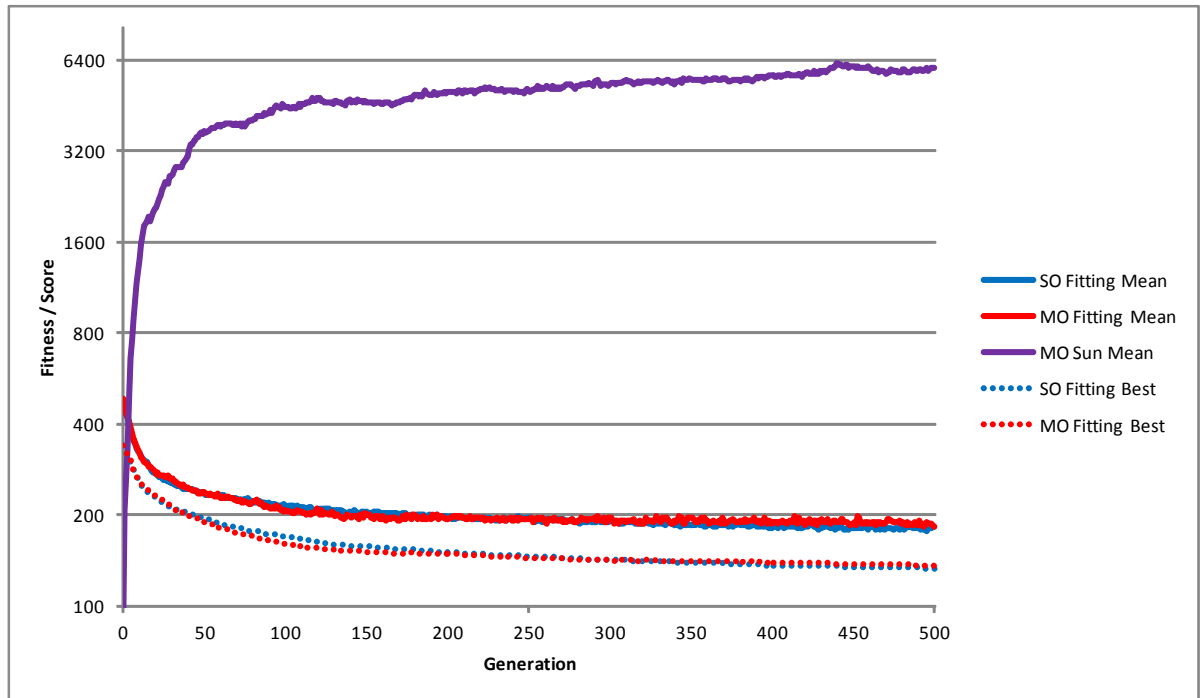
Figure 8.2: Performance graph from single Sun experiment. Graph shows comparison between single and multi-objective experiments over 500 generations from 30 trials.
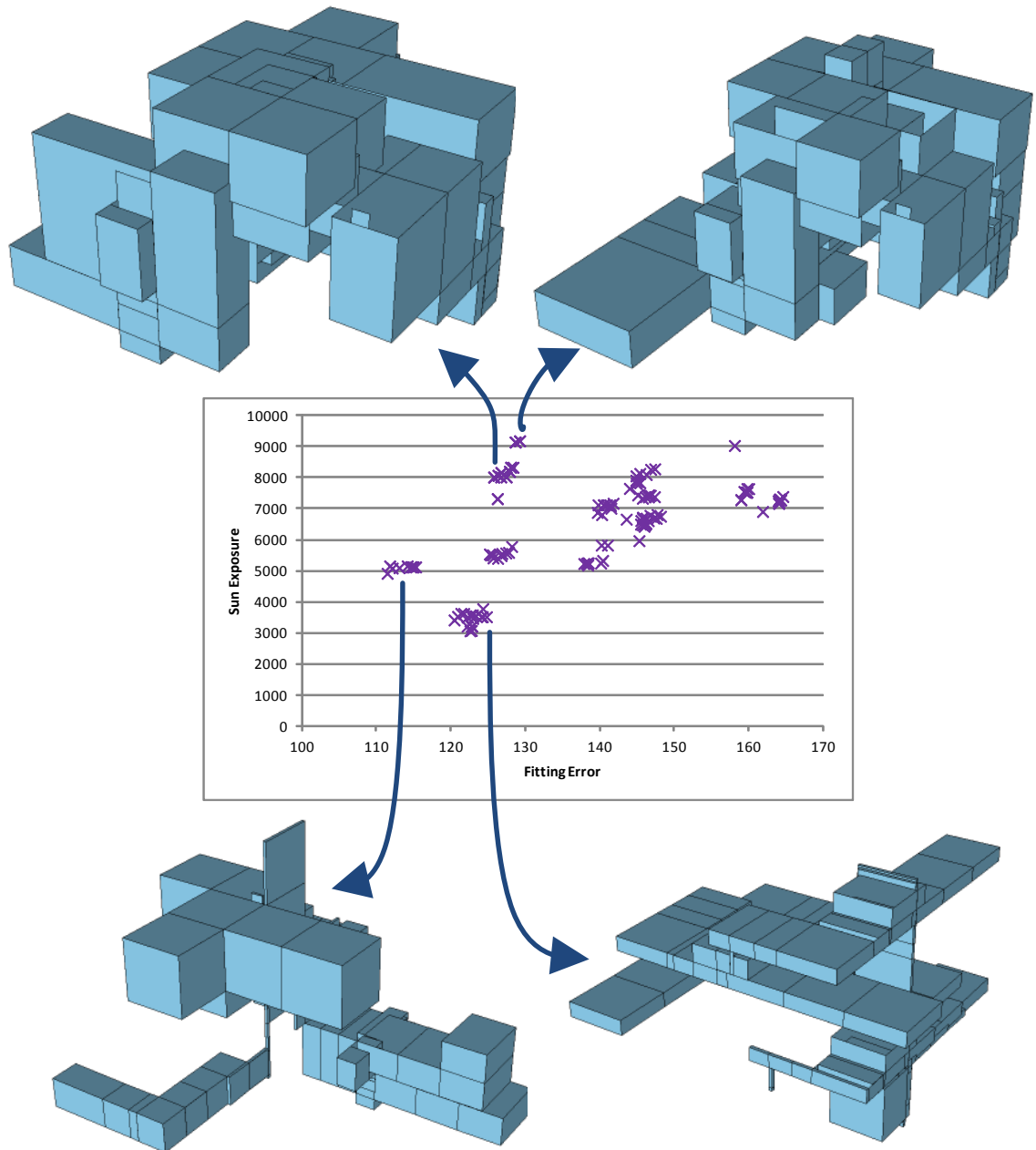
Figure 8.3: Scatter diagram and example individuals from the single Sun experiment. Individuals above diagram have high Sun exposure scores and those below have lower scores.

8.4 Winter Sun Results

Experiments were performed taking into account the movement of the Sun through the sky during the course of a day. Figure 8.4 displays the evolutionary progress of the Winter sun experiment, showing the trend of average Sun exposure, and average and best form fitting scores. The average of best Sun exposure scores are omitted, due to individuals who obtain very high Sun exposure scores while ignoring the form filling objective. It can be seen that both the form fitting and Sun exposure scores are improving over time. The number of generations was reduced to 100 for this experiment, as the experimental trials were very time consuming, though the scores could have improved given more time.

Table 8.4 contains the final scores from the experimental execution. We can see that while the difference in form fitting scores from the single Sun experiment can be explained due to reduced generations, the Sun exposure score is nearly cut in half. This is likely due to the increased difficulty of the multi-Sun problem.
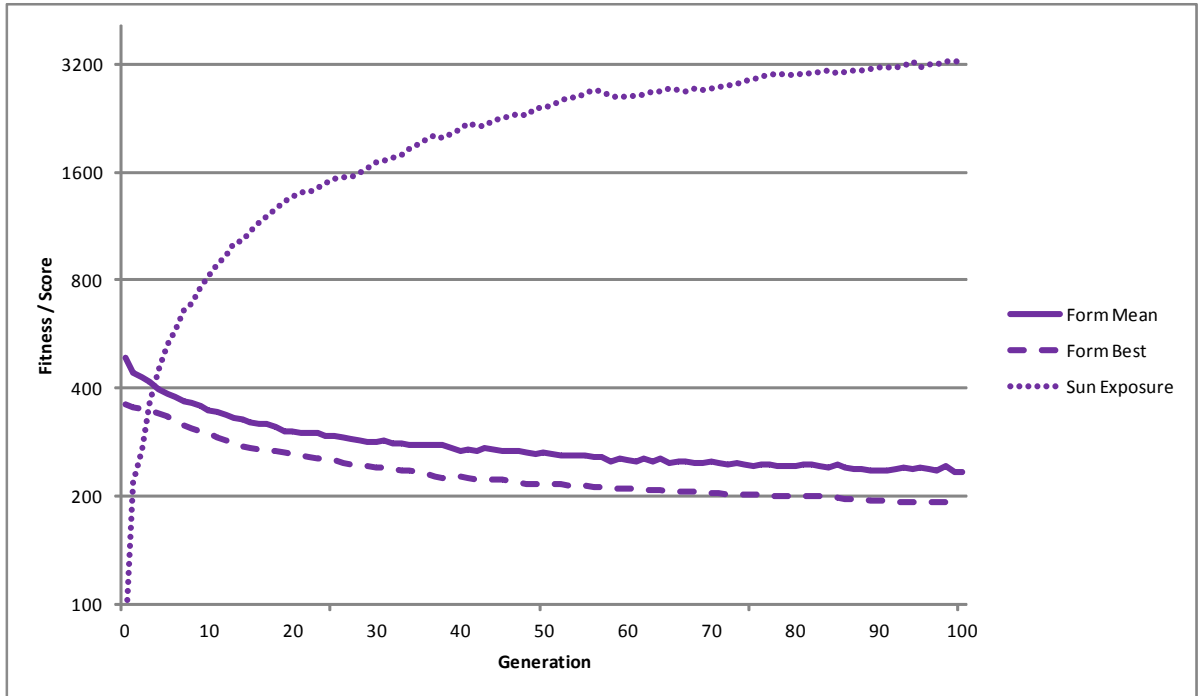

Figure 8.4: Performance graph from winter Sun experiment showing form filling and Sun exposure scores over 100 generations from 15 trials.

Images of two of the best results are seen in Figure 8.5 and 8.6. The images display the average intensity of light from all Sun positions using a greyscale, where white indicates high exposure and

Table 8.4: Average and best fitness scores of the final population from the winter Sun experiment over 15 trials.

Type	Form Fitting		Sun Exposure
	Average	Best	Average
Winter	233.2	191.9	3272.4

black indicates no exposure. These individuals are quite interesting, in that they contain a number of gaps which allow more light to get into the interior of the model. These results resemble that of Watanabe's Sun God city, where form was generated using rule-based logic [46]. Figure 8.7 displays two individuals which received poor Sun exposure scores for comparison. One did poorly due to not having sufficient surface area and the other due to obstructing itself from the Sun, resulting in large amounts of shadow.

The effect of noise on the rendered images is due to overlapping surfaces. It could be good practice to add an objective to minimize surface area in future experiments, to reduce the number of overlapping and hidden surfaces.
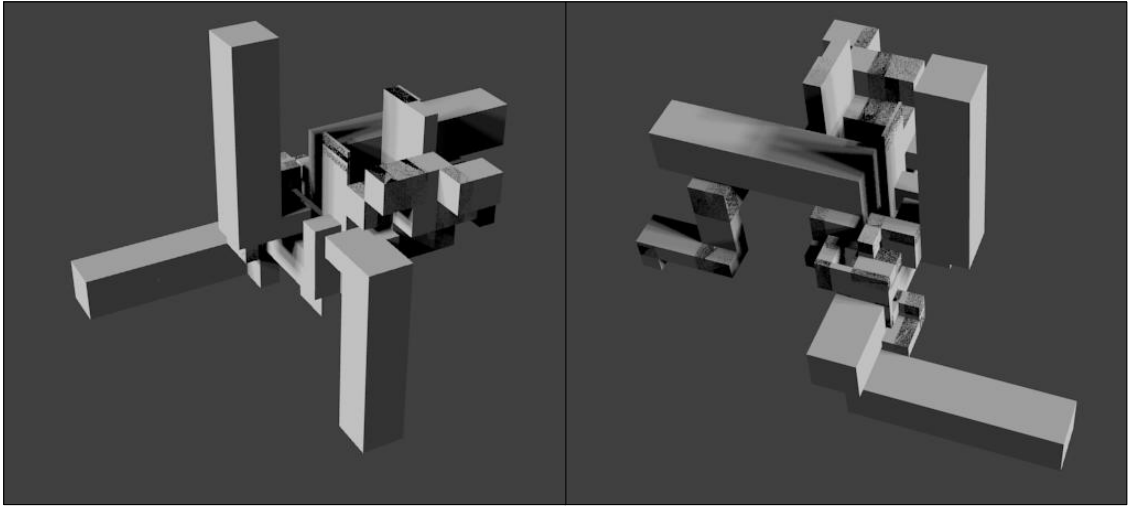
Figure 8.5: Individual with good Sun exposure in winter experiment.

8.5   Shadow City Results

Experiments were executed, using a set of volumes to block Sun exposure to the evolved models, representing other buildings in a city. A surface area minimization objective was added to these experiments, as per observations from the Winter Sun experiments. The evolutionary performance
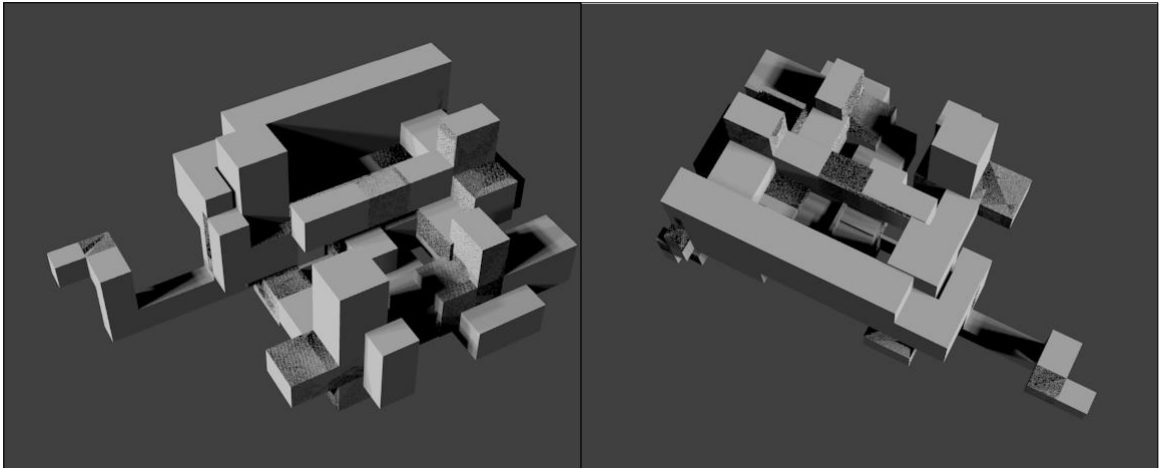
Figure 8.6: Individual with good form fitting in winter experiment.
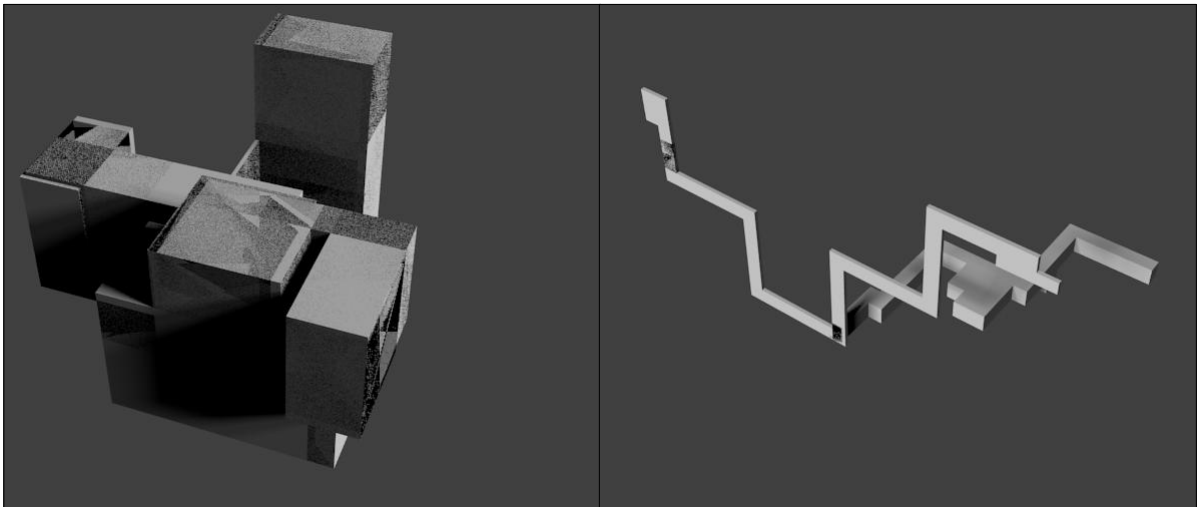
Figure 8.7: Examples of two poor individuals from the winter experiment.

of the experimental trials can be seen in Figure 8.8. Table 8.5 contains the scores from the final population, averaged over the trial runs. In comparison to the Winter Sun experiments, the form fitting has not performed quite as well and the Sun exposure amount is reduced by about a third due to the blocking entities.
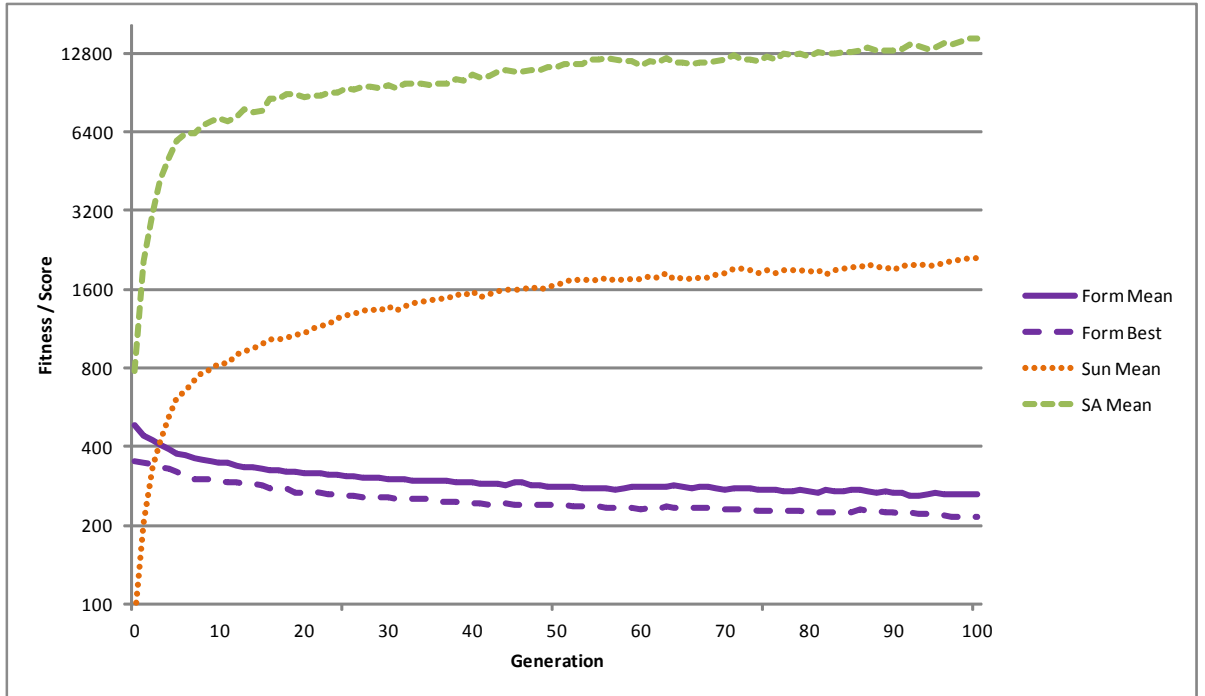
Figure 8.8: Shadow City Performance.

Table 8.5: Average and best fitness from the shadow city experiment.

Type	Form Fitting		Sun Exposure	Surface Area
	Average	Best	Average	Average
Winter	262.2	216.9	2104.5	14525.4

Figure 8.9 contains images of the best result found in these experiments. We can see that the design structure has grown out sideways in both directions to capture more light. It has also grown at a distance from the blocking objects. Figure 8.10 and Figure 8.11 present images of two other strong individuals.
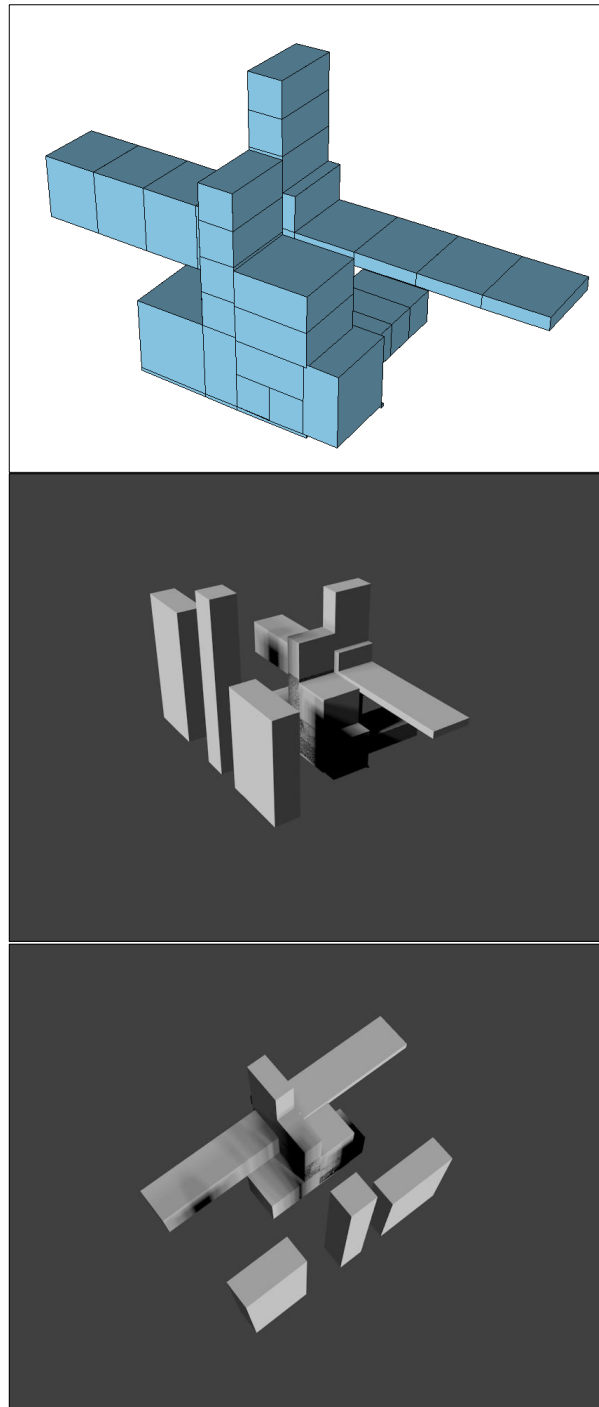
Figure 8.9: An example individual from shadow experiment.
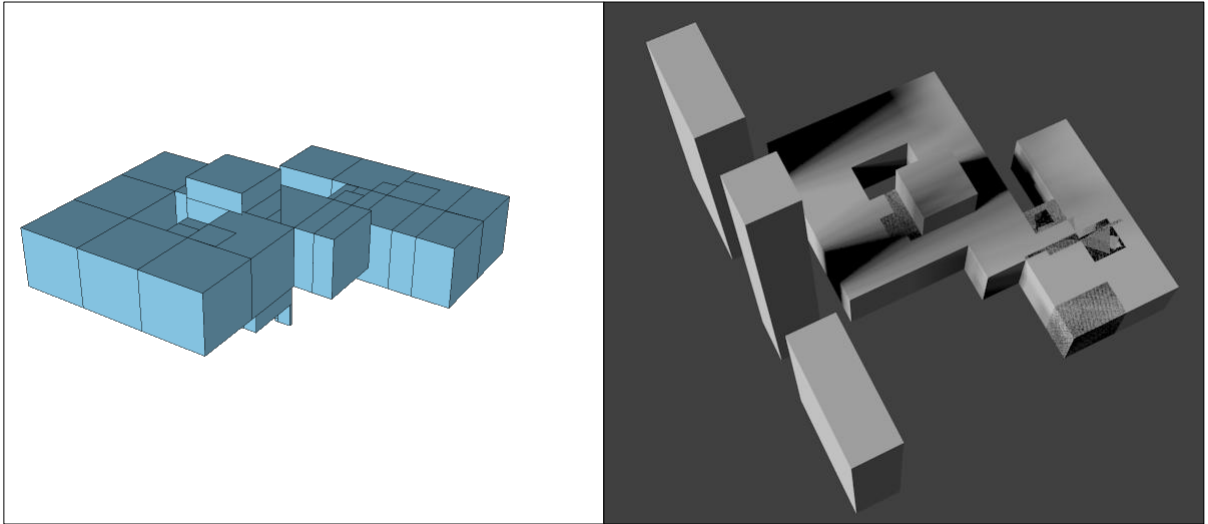
Figure 8.10: An example individual from shadow experiment.
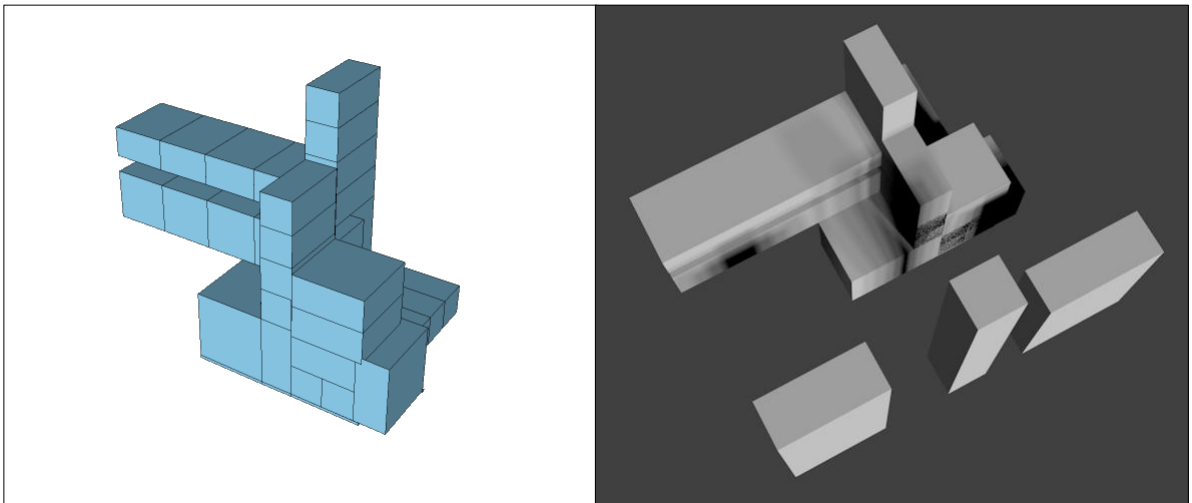
Figure 8.11: An example individual from shadow experiment.

8.6 Conclusions

Moving from previous chapters in which we confirmed the strengths of the MRH representation and the ability of the system to generate form to fill targets, we now focused on applying architecturally related objectives for form generation. We designed a series of new problems, to understand how the system would react to the new objectives and to show that the system is capable of scaling to these more difficult problems.

In the single Sun experiment, we compared the results with the single objective form filling experiment and saw that the addition of the second objective did not alter the performance of first objective. This is an important feature, as the addition of architectural objectives only minimally alters form from its target.

The sunlight exposure problem was then expanded to account for the transit of the Sun across the sky. Example building envelopes were shared, showing averages of light intensity exposure. These design models were compelling, to see that there was a high level of detail.

We then added in some extra buildings external to the generated architecture to simulate the construction of a building in a dense suburban lot.

This chapter successfully showed that genetic programming with generative representations is capable of evolving designs to meet complex objectives. The experiments in this chapter were rather limited in that the Cuboid design language can only generate forms that have six unique face normals. Moving forward it would be beneficial to look at other languages.

Chapter 9

Miscellaneous Runs

This chapter contains experimental details on various sorts of problems. Many questions arose during the development of the system, as well as after witnessing early experimental results. Here, we will address some of those ideas and attempt to explore the capabilities of the system.

9.1 Highly Recursive

With the introduction of the new generative representation parameters, comes a requirement to experiment on those parameters. Many of the parameter values chosen in previous chapters were based on empirical study. In an attempt to improve the features of self-similarity in the generated designs, we will experiment with reducing the procedure amounts and increase the recursive rewrite depth.

9.1.1 Experimental Setup

Table 9.1 contains the GP and encoding parameters used in these experiments. Experiments will be executed on both the small arch and ten random point problems, and will be compared to results from previous chapters. Parameters were set to match with the corresponding runs from previous chapters and can be seen in Table 6.1 and Table 7.1. The procedure amounts were dropped from seven to two, and the rewrite depth was increased from four to six.

9.1.2 Results

Fitness performance over the course of the evolutionary runs can be seen in Figure 9.1 and Figure 9.2. Here we can see that average of best scores performed reasonably well, considering the procedure amount was reduced to two. The average scores for the highly recursive experiments fluctuated

Table 9.1: GP And Encoding Parameters for highly recursive experiments.

Parameter	Value	GR Parameter	Value
Generations	500	**Procedure Amount**	2
Population Size	100	**Rewrite Depth**	6
Crossover Rate	90%	**Condition Pairs**	2
Tournament Size	4	**Design Language**	Growth, P-Cuboid
Elite Count	1	**Symbol Arity**	1
Experimental Trials	30	**Seed Object**	Cube
			(1x1x1 units)

heavily, showing that highly recursive grammars can be destructive during reproduction operators. The final scores of the experiments are shared in Table 9.2, comparing results to those of the previous chapters.

The two best models from each experiment are shown in Figure 9.3. Images of models from low recursive experiments are presented in Figure 6.3 and Figure 7.4. In these examples we can see patterns and examples of self-similarity, showing that recursion is an important factor in generating these aesthetic attributes.

It can be seen from these experiments, that there is a requirement to examine all of the parameters associated with the generative representations in future work.

Table 9.2: Average and best fitness from the highly recursive experiment.

Highly Recursive	Small Arch		Ten Points	
	Average	Best	Average	Best
Yes	271.6	32.9	278.2	143.0
No	49.8	16.4	182.4	133.3

9.2 Skyscrapers

After experimenting with the various features of generative representations and fitness evaluation methods for generating form, we decided to experiment with the generation of models which resemble skyscrapers. This is meant to show that the goals of the system can be altered to generate designs in the context of the user's wishes. Experimentation will take place on attempting to generate forms that resembles skyscrapers, then adding objectives of Sun exposure.

9.2.1 Experimental Setup

We will begin by first only evaluating form, next adding in the Sun exposure fitness and finally we will take into account the locations of the Sun in both Summer and Winter. This problem is
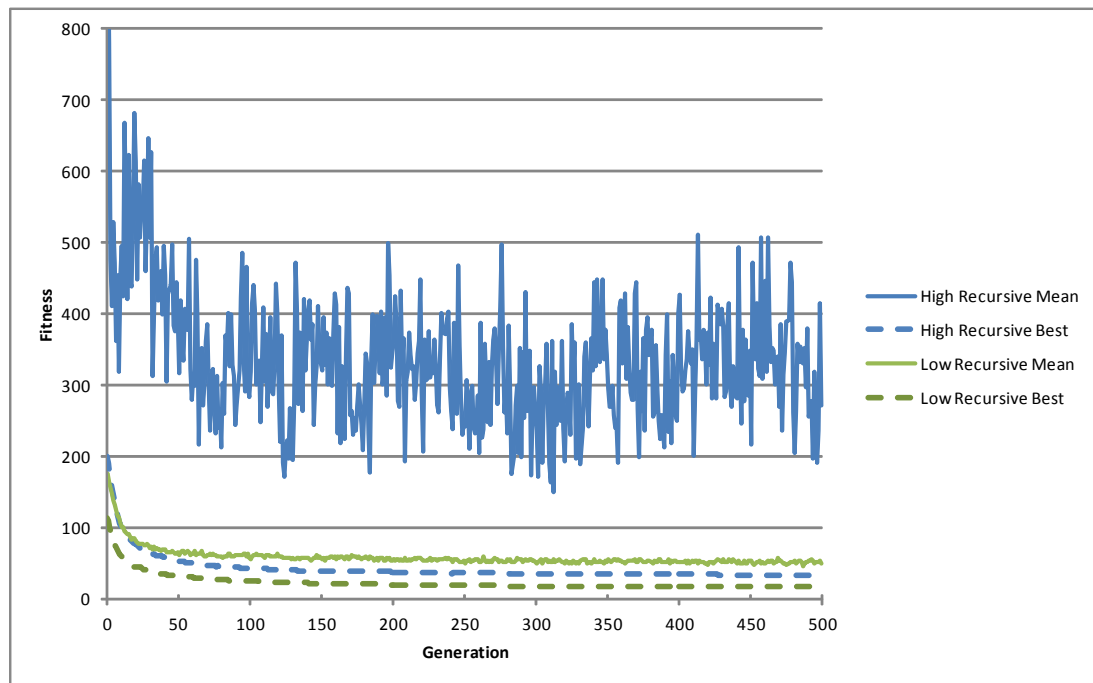
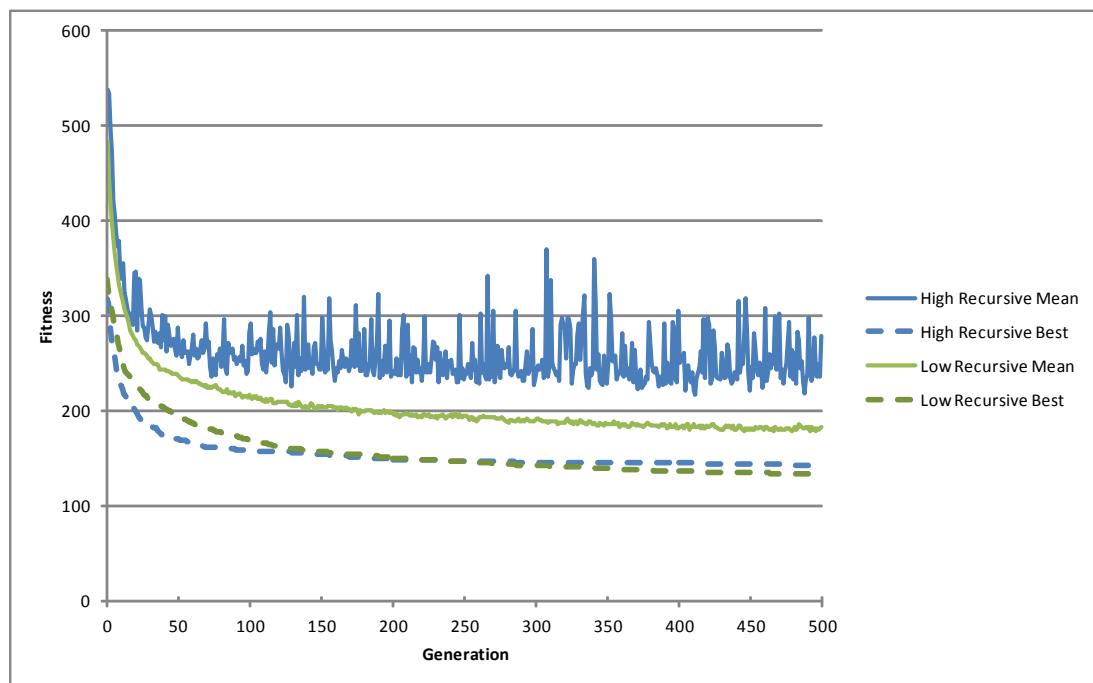Figure 9.1: Evolutionary progress of highly recursive small arch problem.

Figure 9.2: Evolutionary progress of highly recursive ten random points problem.
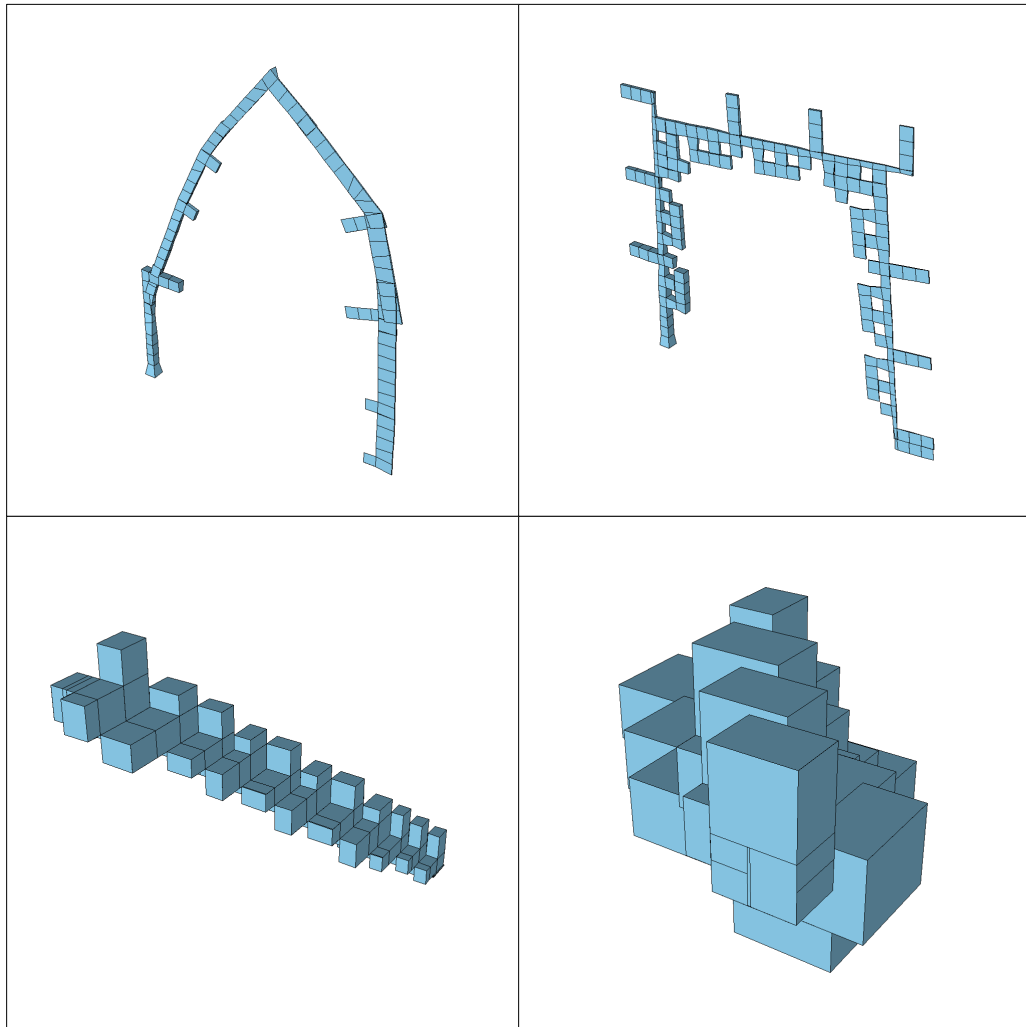
Figure 9.3: Best individuals from highly recursive experiments.

designed to be a conflicting multi-objective problem, where the Sun exposure in the Summer is to be minimized, and the Winter exposure to be maximize. This is a simulation of an architectural problem in form generation called the passive solar problem. A new set of form filling targets were created and they are shown in the appendix in Table A.3.

9.2.2 Results

Table 9.3 contains the final results from the three experiments. We can see that the form fitting scores are all within the same range. When comparing the Winter Sun and passive solar scores, it is seen that the addition of minimizing Summer Sun has impacted the Sun exposure during the Winter heavily.

Examples of individuals can be seen in Figures 9.4 through 9.6. The first figure shows some samples from the skyscraper experiment without influence from the sun. Some interesting building envelopes begins to appear which do resemble skyscrapers. In the second figure, we can begin to see the effects of the Sun exposure and how the form will tilt its surfaces towards the Sun. In the passive solar trials, we begin to see overhanging and sloping surfaces, exposing more surface area towards the Winter Sun.

Table 9.3: Average and best fitness values from the skyscraper experiments.

Experiment	Form Fitting		Winter Sun	Summer Sun
	Average	Best	Average	Average
No Sun	64.1	43.3		
Winter Sun	72.2	50.1	688.0	
Passive Solar	68.3	58.9	301.6	233.5

9.3 Conclusions

We examined a variety of problems and experiments in this chapter. We showed that individuals with low procedure amounts and high recursive depth can still generate reasonable results, with features of self-similarity, though average values tend to fluxuate heavily. There is a need to look more deeply into how these parameters play a role.

The GP system was then asked to generate models which resembled skyscrapers, and experimented with Winter and Summer Sunlight. The system was successful in creating a variety of fascinating models. We witnessed how the Sun exposure evaluations tend to change the shape of the design objects. This exemplified that the system is capable of generating three-dimensional models with a wide variety of styles and appearances.
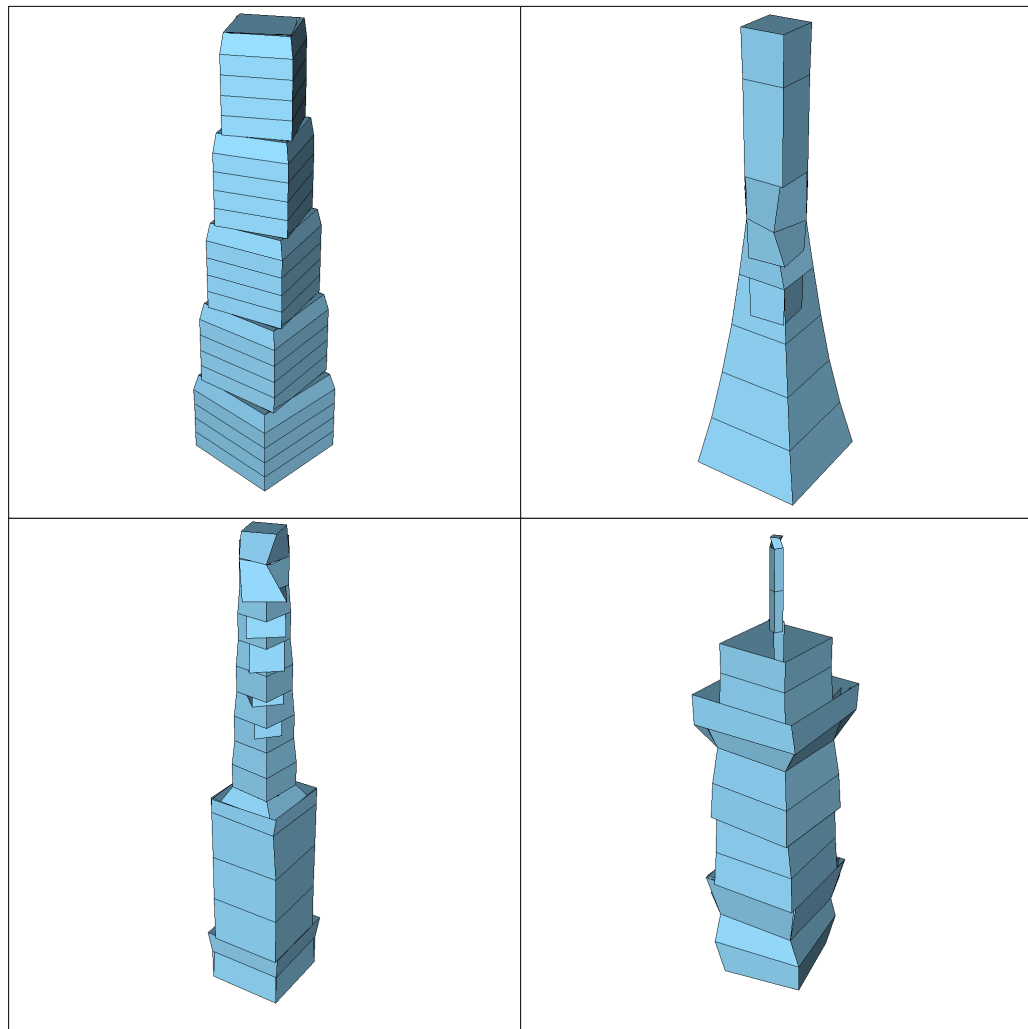
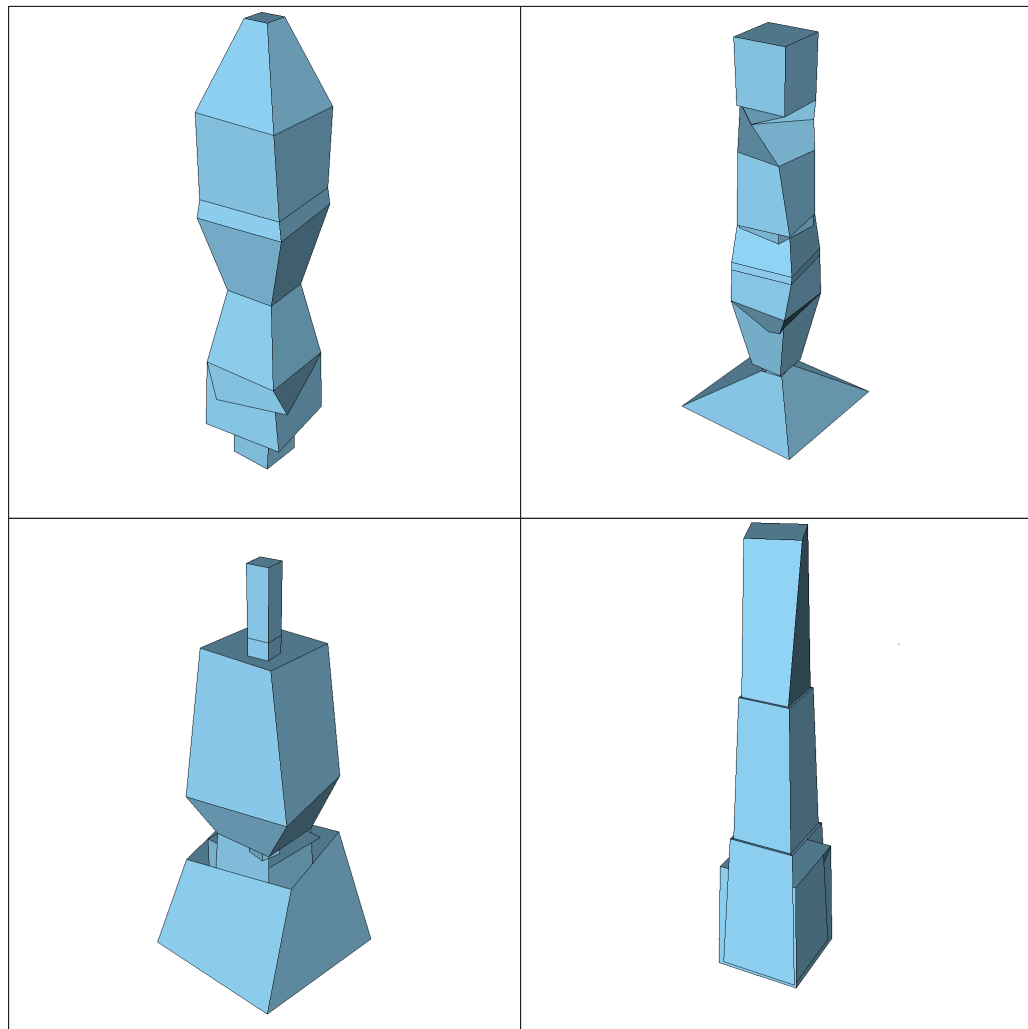Figure 9.4: Example individuals from the simple skyscraper form experiment.

Figure 9.5: Example individuals from the skyscrapers with Winter Sun objective.
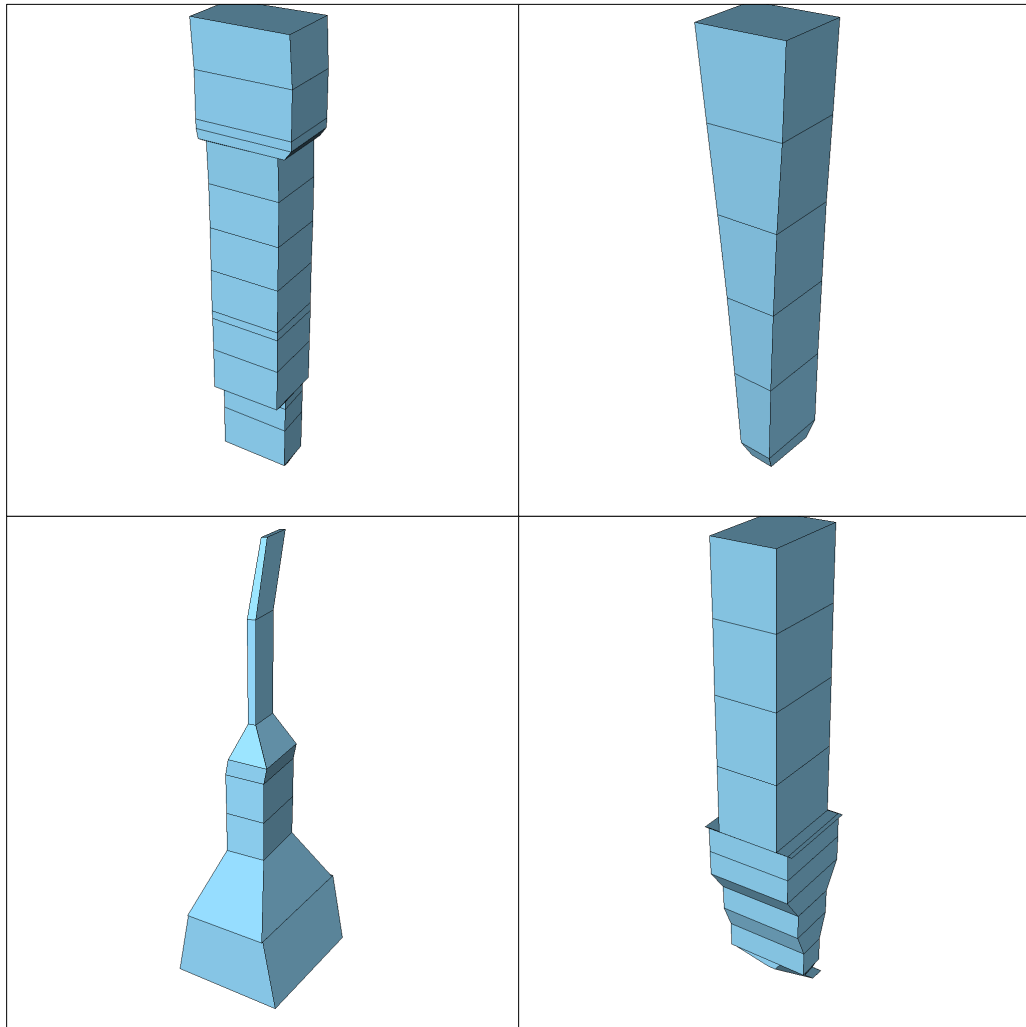
Figure 9.6: Example individuals from the passive solar trials.

Chapter 10

Conclusions and Future Work

This chapter provides a summary of this thesis and the most important findings of this research. It also contains suggestions for what can be done to further extend and enhance this work, through improving on the developed system, design languages or experimentation.

10.1   Conclusions

In this research, we set out to accomplish three major goals. The first goal was to examine and validate the use of generative representations in an evolutionary design problem. The second was to apply the use of generative representations towards an architectural design problem. Finally, the third was to design and examine various design languages for the automatic generation of three-dimensional models.

A genetic programming system was developed which encodes individuals as generative representations. This system was applied to the creation of three-dimensional design models using an array of design languages and fitness functions. Hornby's work was expanded on, by examining the evolution of design structures in a non-voxel space, and applying it to an architectural perspective.

Multiple sets of experiments were carried out in this research. In chapter six, we performed a set of simple experiments on a problem we designed called the arch problem. Here, we compared the five representations on a very simple evolutionary design problem and proved three important points. We showed that the simple representation is not sufficiently scalable in evolutionary design problems of growing complexity. We also showed that the form filling fitness function is a strong method for guiding an evolutionary design system to create geometry. Lastly, it was seen that the developed system was capable of automatically generating a diverse set of novel and inspirational design models.

In chapter seven, a more difficult version of the form filling problem was presented. This problem

used a set of random points to guide the evolution of form. There, we showed that the features of modularity, regularity and hierarchy are extremely important for scalability in evolutionary design problems. These results were consistent with those of Hornby, though minor differences arose in the measures of complexity. We found that high values in the complexity metrics did not equate to better fitness, but that individuals with good metric scores would perform better more often. The measurements of AIC, modularity, reuse and hierarchy were very useful in displaying how the features of complexity enabled higher quality solutions. Using the parametric cuboid design language, a variety of conceptual building envelopes were generated with detailed geometry. These rough forms guided by random target points could potentially be refined using architecturally related fitness evaluations.

In chapter eight we created an array of Sun related experiments. This showed that the proposed system could generate design models that are applicable to an assortment of architectural problems. It was also shown that the Sun exposure fitness did not impact the form filling scores, proving that the system can scale to handle more objectives.

A collection of various experiments was designed and executed in chapter nine. These experiments were performed to generate subjectively aesthetic models and demonstrate the capabilities of the system.

We showed that the classic GP encoding is not sufficiently scalable in complex design problems, validating the use of generative representations, and the features of modularity, reuse and hierarchy. Once the system was capable of generating diverse groups of three-dimensional form, we showed that new objectives could be added to direct the generated form towards architectural problems. Finally, we experimented with various design languages and fitness evaluations, in an attempt to create visually pleasing architectural designs.

This work only begins to scratch the surface of what GP and generative representations are capable of providing, with more refined design languages and evaluation measures, we will be able to provide human-competitive results in artificial architecture.

10.2   Future Work

In this work we were able to successfully design and develop a system capable of generating three-dimensional design models. We showed that generative representations are able to scale to varying levels of complexity in evolutionary design and architectural problems. Though the system was very successful, there are a number of ways in which it could be enhanced to generate even better results. To enhance this research, one could improve on the generative representations, design languages, evaluation functions, or perform continued experimentation.

The encoding of the generative representations could have been improved in a number of ways. In this system, the design language arity was fixed to a set number for all the design operators. A

better approach would allow each of the design operators to have its own arity. The conditional statements were also very limited, only allowing the comparison of a passed parameter to a constant value. This could be extended to allow for more types of conditions.

Allowing access to internal properties of the design model would be beneficial. For example, we could allow a conditional to perform a check on the design model's height before extruding upwards any further. There are a number of parameters that could be exposed and experimented with.

A number of new design languages could be explored. A removal language, that begins with a block and removes components could be interesting. A hybrid language which mixes the growth and removal languages could also have intriguing results.

The form fitting evaluation could be further extended, to include more complicated targets and boundaries. This could include volumes or complicated meshes. The boundaries could restrict growth entirely, or deduct scores for leaving them.

Aside from sun exposure, there are a number of other architectural factors that could be examined. This could include structural integrity, examining the ability to stand against gravity and resistance from the wind. Materials and the costs associated could also have interesting results, such as windows, concrete or steel. There is a growing trend in the architectural community to begin examining energy efficiency in designs. The system presented here could be tailored towards those problems.

With the creation of new parameters for the generative representations, comes the need to experiment with those parameters to gain a better understanding of how they work.

The ground work has been laid and there is proof that these systems are a step in the right direction. There is potential to create systems with remarkable outcomes with some more work.

Bibliography

[1] Peter J. Bentley and Jonathan P. Wakefield. Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 231–240. Springer-Verlag, January 1998.

[2] Steve Bergen. Automatic structure generation using genetic programming and fractal geometry. Master's thesis, Brock University, 2011.

[3] Steve R. Bergen. Evolving stylized images using a user-interactive genetic algorithm. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2745–2752, New York, NY, USA, 2009. ACM.

[4] Craig Caldwell and Victor S. Johnston. Tracking a Criminal Suspect through "Face-Space" with a Genetic Algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithm*, pages 416–421. Morgan Kaufmann Publisher, July 1991.

[5] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[6] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Corrado Coia and Brian J. Ross. Automatic evolution of conceptual building architectures. In *IEEE Congress on Evolutionary Computation*, pages 1140–1147, 2011.

[8] David W. Corne and Peter J. Bentley. *Creative Evolutionary Systems (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, 1st edition, July 2001.

[9] Robert W. J. Flack and Brian J. Ross. Evolution of architectural floor plans. In *EvoApplications (2)*, pages 313–322, 2011.

[10] J. Frazer. *An evolutionary architecture.* Themes Series. Architectural Association, 1995.

[11] John Frazer, Julia Frazer, Liu Xiyu, Tang Mingxi, and Patrick Janssen. Generative and Evolutionary Techniques for Building Envelope Design, 2002.

[12] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1 edition, January 1989.

[13] Jan Halatsch, Antje Kunze, and Gerhard Schmitt. Using shape grammars for master planning. *Design Computing and Cognition 08 Proceedings of the Thrid International Conference on Design Computing and Cognition*, pages 655–673, 2008.

[14] Martin Hemberg, Una-May O'Reilly, Achim Menges, Katrin Jonas, Michel Gonçalves, and Steven Fuchs. Genr8: Architects' experience with an emergent design tool. In *The Art of Artificial Evolution*, pages 167–188, 2008.

[15] G. S. Hornby. Functional Scalability through Generative Representations: the Evolution of Table Designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, July 2004.

[16] Gregory S. Hornby. *Generative representations for evolutionary design automation.* PhD thesis, Waltham, MA, USA, 2003.

[17] Gregory S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1729–1736, New York, NY, USA, 2005. ACM.

[18] Gregory S. Hornby. Measuring complexity by measuring structure and organization. In *2007 IEEE Congress on Evolutionary Computation*, pages 2017–2024. IEEE Press, 2007.

[19] Christian Jacob. *Illustrating Evolutionary Computation with Mathematica.* Morgan Kaufmann, 2001.

[20] Rafal Kicinger, Tomasz Arciszewski, and Kenneth De Jong. Evolutionary computation and structural design: A survey of the state-of-the-art. *Comput. Struct.*, 83(23-24):1943–1978, September 2005.

[21] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[22] John R. Koza. *Genetic programming II: automatic discovery of reusable programs.* MIT Press, Cambridge, MA, USA, 1994.

[23] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[24] Aristid Lindenmayer. Adding continuous components to l-systems. In *L Systems, Most of the papers were presented at a conference in Aarhus, Denmark*, pages 53–68, London, UK, UK, 1974. Springer-Verlag.

[25] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 102:1–102:10, New York, NY, USA, 2008. ACM.

[26] S. Luke. ECJ - a java-based evolutionary computation research system. In *http://cs.gmu.edu/eclab/projects/ecj/*.

[27] Penousal Machado and Fernando Graca. Evolutionary pointillist modules: evolving assemblages of 3D objects. In *Evo'08: Proceedings of the 2008 conference on Applications of evolutionary computing*, pages 453–462, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] James McDermott, Niall Griffith, and Michael ONeill. Evolutionary computation applied to sound synthesis. In Juan Romero and Penousal Machado, editors, *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*, pages 81–101. Springer Berlin Heidelberg, 2007.

[29] David J. Montana. Strongly typed genetic programming. *Evol. Comput.*, 3(2):199–230, June 1995.

[30] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, July 2006.

[31] Michael ONeill and Anthony Brabazon. *Evolving a Logo Design Using Lindenmayer Systems, Postscript and Grammatical Evolution*, pages 3788–3794. IEEE Press, 2008.

[32] Michael ONeill, James McDermott, John Mark Swafford, Jonathan Byrne, Erik Hemberg, Anthony Brabazon, Elizabeth Shotton, Ciaran McNally, and Martin Hemberg. Evolutionary design using grammatical evolution and shape grammars: Designing a shelter. *International Journal of Design Engineering*, 3(1):4–24, 2010.

[33] Una-May O'Reilly and Martin Hemberg. Integrating generative growth and evolutionary computation for form exploration. *Genetic Programming and Evolvable Machines*, 8(2):163–186, June 2007. Special issue on developmental systems.

[34] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[35] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[36] Juan Romero and Penousal Machado, editors. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*. Natural Computing Series. Springer Berlin Heidelberg, November 2007.

[37] T. Sakamoto and A. Ferré. *From Control to Design: Parametric/Algorithmic Architecture*. Verb monograph. Actar-D, 2008.

[38] N.A. Salingaros and M.W. Mehaffy. *A Theory Of Architecture*. Umbau-Verlag, 2006.

[39] Karl Sims. Artificial evolution for computer graphics. *Computer Graphics*, pages 319–328, 1991.

[40] Karl Sims. Evolving 3d morphology and behavior by competition. *Artif. Life*, 1:353–372, January 1994.

[41] Michael Smyth and Ernest A. Edmonds. Supporting design through the strategic use of shape grammars. *Knowl.-Based Syst.*, 13(6):385–393, 2000.

[42] G Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, May 1980.

[43] George Stiny. *Shape: Talking about Seeing and Doing*. The MIT Press, 2008.

[44] Kostas Terzidis. *Algorithmic Architecture*, volume 1. Architectural Press, 2006.

[45] Peter von Buelow. *Genetically Engineered Architecture - Design Exploration with Evolutionary Computation*. VDM Verlag, Saarbrücken, Germany, Germany, 2007.

[46] M.S. Watanabe. *Induction Design: A Method for Evolutionary Design*. The IT Revolution in Architecture. Birkhäuser, 2002.

[47] Emily Whiting, John Ochsendorf, and Frédo Durand. Procedural modeling of structurally-sound masonry buildings. *ACM Trans. Graph.*, 28(5):112:1–112:9, December 2009.
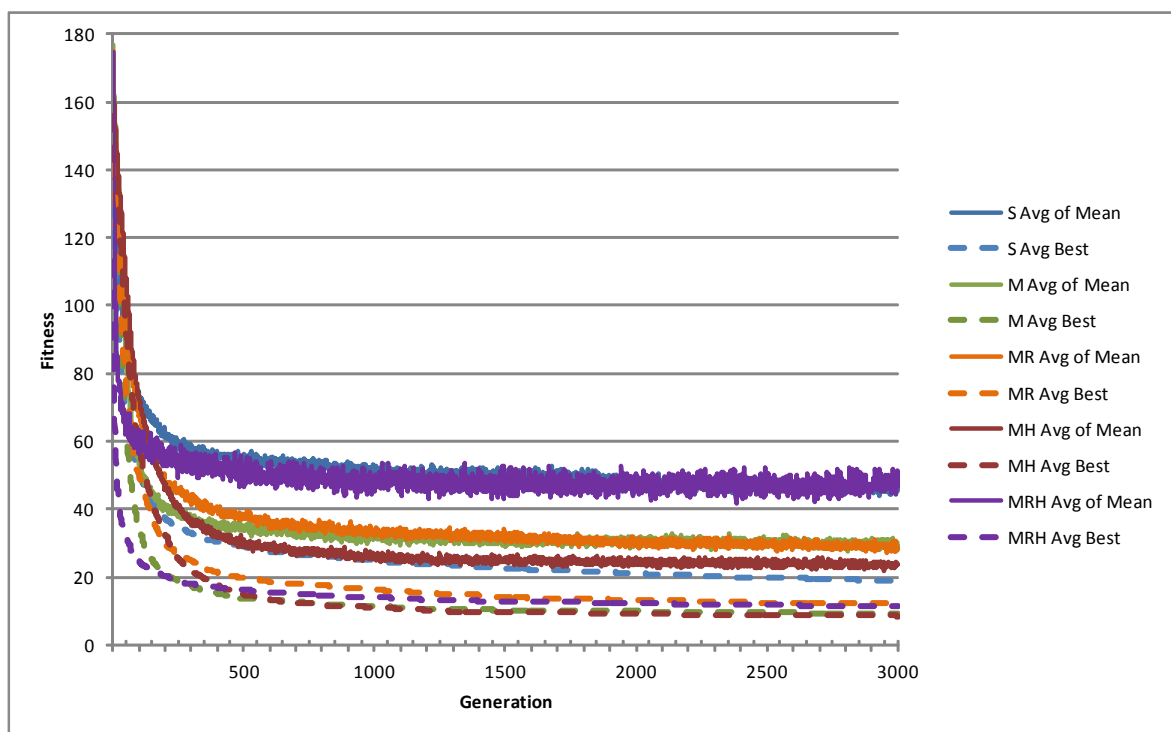
Appendix A

Miscellaneous Results

Figure A.1: Performance graph from the small arch experiment. Displaying the average of mean and best fitness over 3000 generations from 30 experimental trials
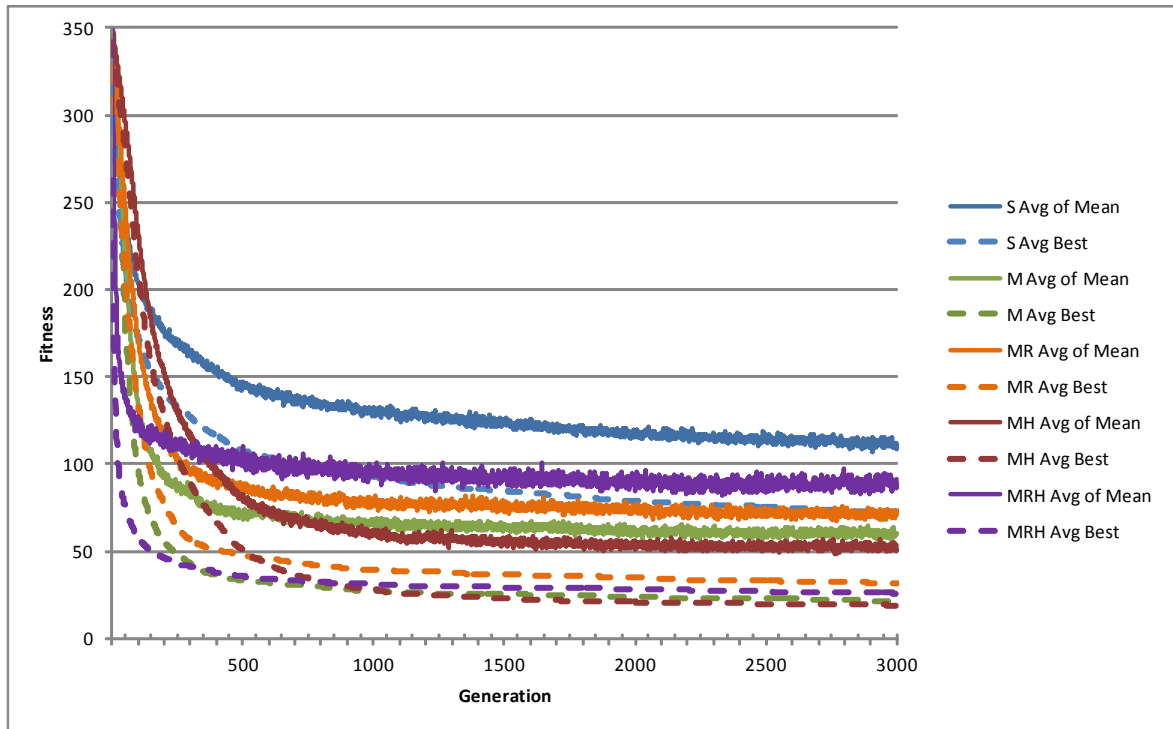
Figure A.2: Performance graph from the medium arch experiment. Displaying the average of mean and best fitness over 3000 generations from 30 experimental trials
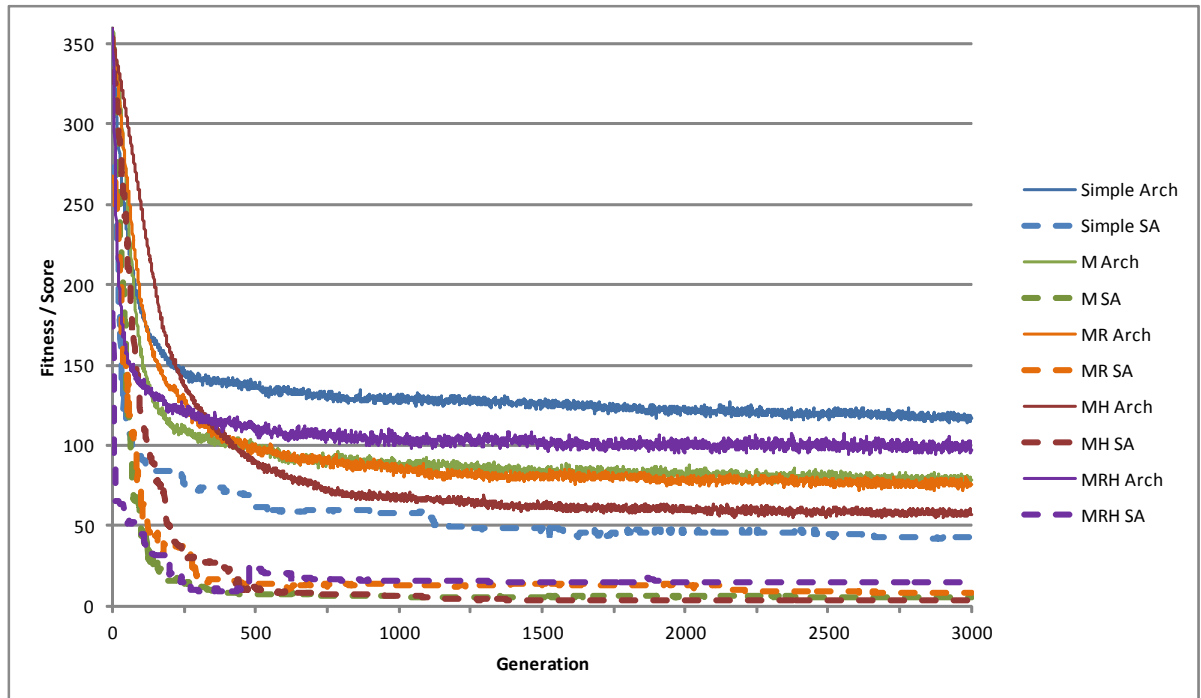
Figure A.3: Performange graph of arch error and surface area on the medium arch problem. Displaying the average of mean over 3000 generations from 30 trials.
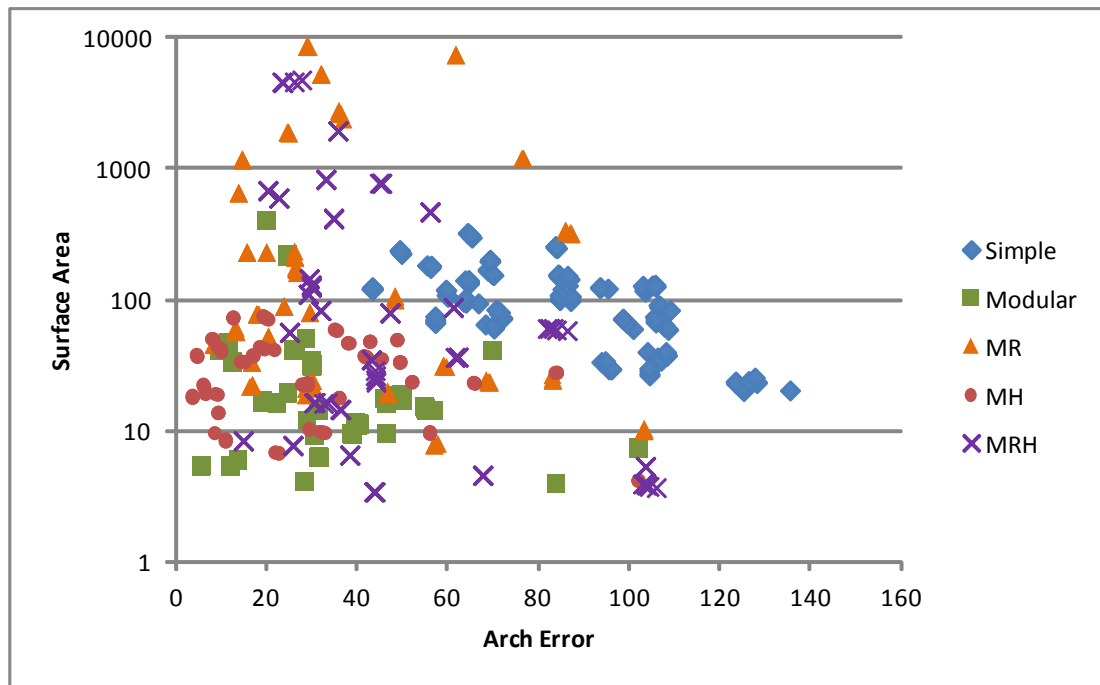
Figure A.4: Distribution of 150 total solutions, showing 30 best of each representation in the arch error versus surface area experiment.

Table A.1: 50 Randomly generated target values

Id	x	y	z	Id	x	y	z
1	47	37	16	**26**	39	27	-19
2	-20	44	38	**27**	32	31	8
3	-18	10	40	**28**	-36	42	-41
4	19	9	-49	**29**	5	35	-19
5	-16	17	-50	**30**	17	33	-1
6	-36	36	-21	**31**	-44	4	27
7	30	2	-24	**32**	20	2	-31
8	10	42	-14	**33**	1	14	-26
9	37	32	22	**34**	-30	12	19
10	-5	47	7	**35**	-13	38	-2
11	-8	23	30	**36**	-4	24	-11
12	-38	6	25	**37**	-42	29	-25
13	36	42	-50	**38**	45	35	50
14	15	45	-41	**39**	8	30	31
15	-1	26	-44	**40**	-26	21	-21
16	-31	33	-40	**41**	23	39	-50
17	3	32	2	**42**	23	44	44
18	-3	24	-47	**43**	-39	24	-10
19	-45	47	8	**44**	-4	48	16
20	-8	13	8	**45**	-44	43	-18
21	3	11	24	**46**	-7	44	-29
22	10	7	5	**47**	3	34	-9
23	27	24	-22	**48**	-37	12	23
24	-49	10	30	**49**	11	24	-50
25	11	35	-19	**50**	-44	44	1

Table A.2: Sunlight vectors

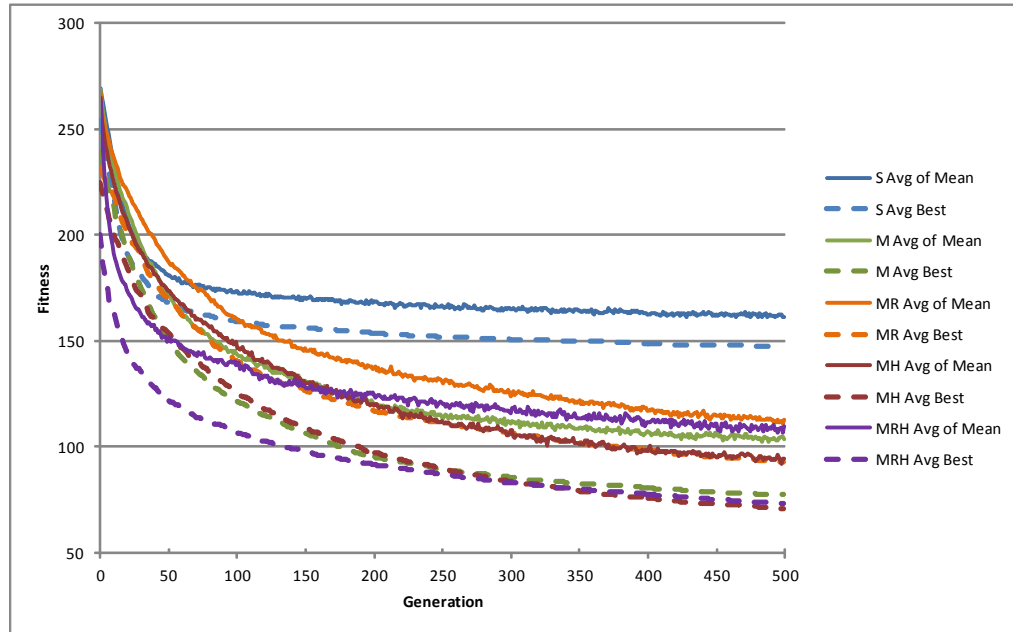|	Summer				Winter		
Id	x	y	z	**Id**	x	y	z
1	-3	5	10	**1**	-3	3	10
2	-2	9	10	**2**	-2	5	10
3	-1	12	10	**3**	-1	7	10
4	0	13	10	**4**	0	8	10
5	1	12	10	**5**	1	7	10
6	2	9	10	**6**	2	5	10
7	3	5	10	**7**	3	3	10

Figure A.5: Five random point performance graph.

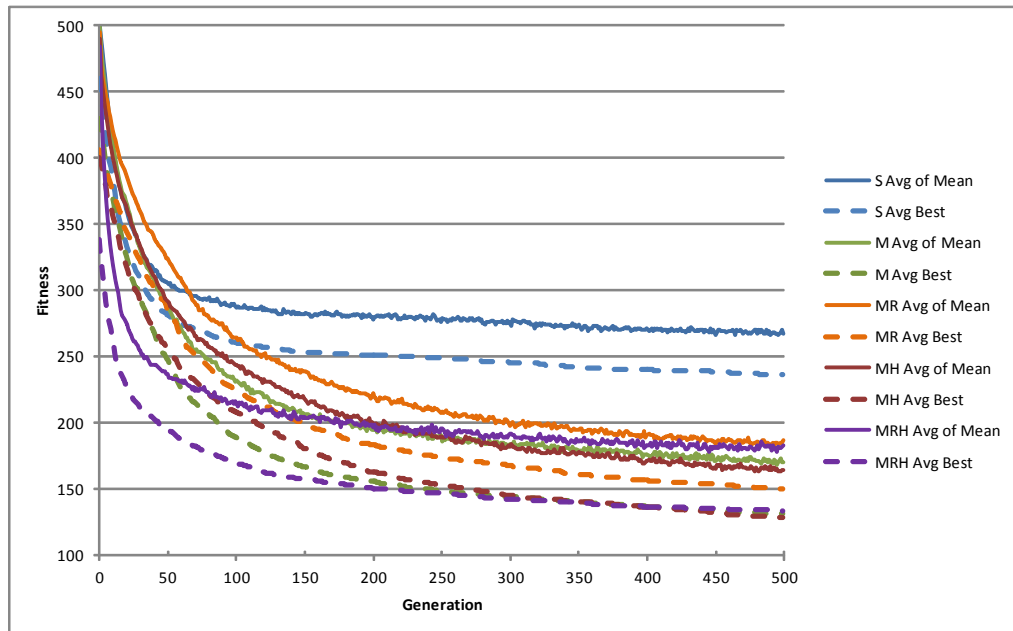
Figure A.6: Ten random point performance graph.
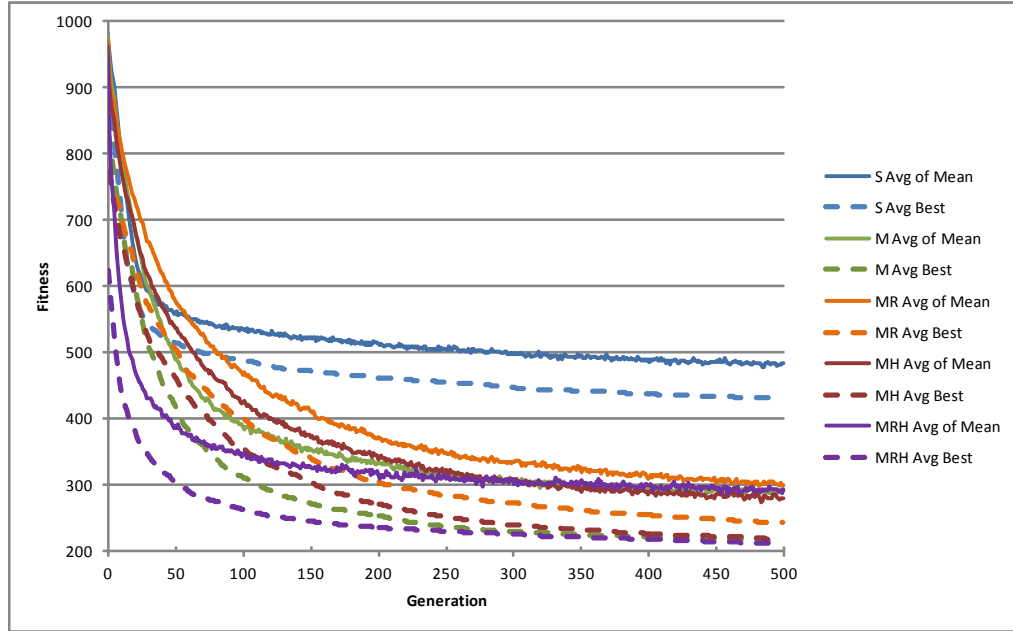
Figure A.7: Twenty random point performance graph.

Table A.3: Skyscraper targets.

Id	x	y	z	Id	x	y	z
1	0	0	0	11	0	20	-5
2	5	0	0	12	0	20	0
3	-5	0	0	13	-5	20	0
4	0	0	5	14	0	20	5
5	0	0	-5	15	-5	30	0
6	5	10	0	16	0	30	0
7	0	10	0	17	0	30	5
8	-5	10	0	18	0	40	0
9	0	10	5	19	0	40	5
10	0	10	-5	20	0	50	0