# Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution

**by**

**Michael O'Neill**

**B.Sc.**

**Supervisor**: Dr. Conor Ryan

**External Examiner**: Prof. Wolfgang Banzhaf

A thesis for the PhD Degree

Submitted to the University of Limerick

2001

# Abstract

We present a novel Evolutionary Automatic Programming system, Grammatical Evolution that is capable of generating programs in an arbitrary language from a binary string.

Grammatical Evolution adopts a genotype to phenotype mapping; the genotype is the raw genetic material, analogous to the DNA of Molecular Biology, and the phenotype the functional program that is generated (the equivalent of proteins in Molecular Biology). Resulting from the genotype-phenotype distinction, and inspired by Molecular Biology, a number of features are introduced that result in benefits for Grammatical Evolution.

We demonstrate Grammatical Evolution's viability on a number of proof of concept problems with performance on a par with, and in some cases superior to Genetic Programming. An analysis of the system is conducted in which we focus on a number of features arising directly from the genotype-phenotype distinction, namely the degenerate genetic code, and the novel, wrapping operator.

We conclude the investigations with an analysis of the effects of the genetic operator of crossover on Grammatical Evolution, before detailing our conclusions and outlining directions for future research.

# Acknowledgements

There are many people that I wish to thank for the support they have given me over the past three and a half years. Firstly, to **Dr. Conor Ryan** my supervisor, colleague and friend, without whom this work would not have been possible. Conors neverending patience, enthusiasm, support, and advice made the whole experience both enjoyable and rewarding.

**J.J. Collins** has been another pillar that I have come to rely on from the very outset of my time in the University of Limerick. Our many conversations about this work and other related areas continue to be a source of inspiration. I must thank him especially for allowing me to use his genetic algorithm code that became the engine behind my experiments.

**Miguel Nicolau** has been of enormous assistance in the preparation of this document, and on many occassions has dropped everything to come to my rescue. Miguel also eased the burden of handcoding the grammar files used in my research with his BNF parser, Gramma, and has provided many other useful scripts to analyse experimental results. His persistence at encouraging me to have a life outside the lab eventually paid off.

Over the past few years I have met a number of people at the various Genetic Programming conferences whose insights have proven invaluable to this work, with the result that I have returned to Ireland with renewed enthusiasm on each occasion. In particular, I would like to thank **Maarten**

**Keijzer**, **Mike Cattolico**, **Forrest Bennet**, **Dr. Bill Langdon**, and **Dr. Adrian Trenaman**. **Norman Paterson**, whom I hadn't met by this time, very kindly provided support when I was replicating his Caching Algorithm experiments by sending me the stream files and grammars that he used.

To the various people with whom I have had the pleasure to share a lab with. To **Jim Buckley** and **Michael English** who took me under their wings when I first arrived in Limerick. **Alan Sheahan** has been an invaluable statistician and played a large role in devising the diversity measure used in my analysis experiments. **Lucia Sheehan** who on a few rare occasions managed to convince me that playing squash was a good idea. **Laur Ivan** came to the rescue of my machine on a number of occasions, and helped write the caching algorithm simulator.

The Department of Computer Science & Information Systems has come to be my second home during this time. To our head of department, **Dr. Donncha O'Maidin** for his support in the later stages of this project with his sparing assignment of teaching loads, and financial assistance that bridged the gap when my original funding source ran dry. To the administrative staff **Carmel Doherty**(RIP), **Gemma Ryan**, **Maria O'Connell**, **Nuala Kitson**, and **Bianca Pluck** for their neverending patience and support. To our technical staff, **Tony Molloy** and **Redmond O'Brien** who always ensured my machines were in good working order. **Prof. Tony Cahill** was a fountain of knowledge who provided many useful insights into Grammatical Evolution, especially from the grammar perspective. I am very grateful to **Dr. Malachy Eaton** for stepping in to present our papers at AROB'2000.

To **Jennifer Willies** and **EvoNet** for their support, especially of the student travel grant kind, that helped me to attend EuroGP'99. **GECCO/AAAI** must also be thanked in this regard with the provision of travel grants that enabled me to attend the annual GP and GECCO conferences, and for running the **Graduate Student Workshops** that provided invaluable feedback.

My family and friends have helped me in ways they will never know. To my parents, **John** and **Jane**, words cannot express the support and encouragement they have given me. To **Fr. Arthur O'Neill** who provided me with a means of transport when I first came to Limerick in the form of my first car. The **Dublin gang** stood by me all this time, even after leaving the confines of the Pale, you all know who you are. Most especially, I must thank **Grainne** for always being there with encouragement and for listening to my endless tales of Genetic Programming.

This work was in part supported by **Forbairt**.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview

We present Grammatical Evolution (GE), an evolutionary algorithm that can evolve complete programs in an arbitrary language using a variable-length binary string.

We identify a set of features from Molecular Biology that confer a number of desirable characteristics onto Evolutionary Algorithms (EAs), and in particular Evolutionary Automatic Programming.

Through the adoption of a genotype-phenotype distinction, one of Molecular Biology's most defining features, it is possible to facilitate the existence of these characteristics. These include, firstly, a separation of the search and solution spaces that allow the EA search engine to be a plug-in component of the system, facilitating the exploitation of advances in EAs by GE. Secondly, this allows the evolution of programs in an arbitrary language with the representation of a program's syntax in the form of a grammar definition.

Thirdly, the existence of a degenerate genetic code is enabled, giving a many-to-one mapping, that allows the exploitation of neutral evolution to enhance the search efficiency of the EA. Fourthly, we can adopt the use of a wrapping operator that allows the reuse of genetic material during a genotype-phenotype mapping process.

## 1.2 Contributions of Thesis

- Automatic generation of programs in an arbitrary language. The mapping process arising from the genotype-phenotype distinction allows the use of a grammar as a plug-in component. The grammar which is used in a generative process to produce the output code can represent the syntax of any language.

- A program representation coupled to a mapping process that enables the search algorithm to be an independent entity of the system. Thus, any search paradigm can be adopted; in particular, the system was designed such that it could adopt advances made in the field of Evolutionary Computation.

- The use of a degenerate genetic code, resulting in a novel many-to-one mapping from genotype to phenotype.

- The introduction of the wrapping operator that can reuse genetic material during a genotype-phenotype mapping process.

- The identification of a number of desirable features that can result in advantages for Evolutionary Automatic Programming systems and Evolutionary Algorithms with the use of a genotype-phenotype distinction.

- An analysis of the Grammatical Evolution system focusing on the degenerate genetic code, wrapping operator, and crossover.

## 1.3 Outline of Thesis

There now follows an outline of each chapter.

**Chapter 2 : Survey of Automatic Programming**

Chapter 2 contains a survey of the field of Automatic Programming, concentrating on Evolutionary Automatic Programming.

**Chapter 3 : Lessons from Molecular Biology**

This chapter details lessons that can be learned from the field of Molecular Biology, describing how these might be used to improve various aspects of Evolutionary Automatic Programming systems, and Evolutionary Algorithms in general. For example, by adopting a genotype-phenotype distinction it is possible to generate programs in an arbitrary language, and to preserve genetic diversity within the evolving populations thus acting as a preventative measure against premature convergence on local optima.

**Chapter 4 : Grammatical Evolution**

A description of the Grammatical Evolution system is given. Grammatical Evolution takes note of the findings in Chapters 2 & 3 and builds upon these to develop a system that is capable of generating programs in an arbitrary language.

**Chapter 5 : Proof of Concept**

Grammatical Evolution is applied to a number of benchmark problems, and its success compared to Genetic Programming. The problems investigated are a symbolic regression problem, a symbolic integration problem, the Santa Fe trail, and the evolution of Caching Algorithms.

**Chapter 6 : Analysis of Grammatical Evolution**

An in depth analysis of Grammatical Evolution is conducted. In particular, attention is focused upon the degenerate genetic code and the wrapping operator. It is shown that these features play an important role in GE's success.

**Chapter 7 : Crossover in Grammatical Evolution**

Crossover in Grammatical Evolution is examined. It is found that crossover has some useful characteristics that the system gets for free with the program representation adopted, such as an exchange of on average 50% of the genetic material.

**Chapter 8 : Conclusions & Future Work**

The thesis concludes with an overview, conclusions, and outlines possible directions for future research.

# Chapter 2

# Survey of Evolutionary Automatic Programming

## 2.1 Introduction

Before a survey of Evolutionary Automatic Programming, it is worth considering what we mean by the term Automatic Programming. Since the inception of a programmable computing device, researchers have sought to achieve Automatic Programming. The meaning of this term, however, has changed over time, as expectations are relative to the current technologies available [Rich & Waters, 1988].

In the early days of computing Automatic Programming referred to assemblers, that is, programs that would automatically generate the machine code from a program written in assembly language. At this time, an assembler relieved a huge burden for a programmer, automating the task of physically writing the binary codes that made up a machine code program. As time progressed, compilers for second generation programming languages such as Fortran were considered Automatic Programming. Following along the lines of the development of more sophisticated compilers, came the third and fourth generation programming languages, and in the 1990's much work has been done on the automatic parallelisation of code, see for example [Foster, 1991] [Lovely, 1992] [Blume & Eigenmann, 1992] [Ryan, 1999].

Today, Automatic Programming is in the realm of Intelligent Systems, and is perhaps best described by Arthur Samuel [Samuel, 1959] when he said

> "Tell the computer what to do, not how to do it."

In other words, we wish to be able to automatically generate a program from scratch given a high-level problem description. For this definition of Automatic Programming we must adopt the technologies of machine learning in order to achieve this goal. To date, the most successful approaches in this direction are Genetic Programming, an Evolutionary Automatic Programming approach, and Inductive Logic Programming.

Inductive Logic Programming (ILP)[Muggleton, 1992] is an approach that combines logic programming and machine learning and typically adopts the Prolog language. The ILP approach has produced systems like FOIL (First Order Inductive Logic) [Quinlan, 1990], a natural extension of ID3, that induces decision trees.

More recently there has been work to combine the best of these two approaches, that is from ILP taking the notion of a logic program and from Evolutionary Automatic Programming the ability to evolve programs to produce a system that can evolve logic programs [Wong & Leung, 2000].

A number of attributes that an automatic programming system should possess have been indentified in [Koza et.al., 1999]. These attributes include:

- the ability to start with a high-level problem description that results in a solution in the form of a computer program,

- the ability to automatically determine the programs size and architecture,
- the ability to automatically organise a group of instructions so that they may be re-used by a program,
- problem-independence,
- scalability to larger versions of the same problem,
- and the capability of producing results that are competitive with those produced by humans.

It has been noted in [Koza et.al., 1999] that an Evolutionary Automatic Programming approach successfully achieves each of the identified attributes.

We will now continue by conducting a review of Evolutionary Automatic Programming as this is the area in which the work of this thesis lies.

## 2.2 Evolutionary Automatic Programming

The phrase Evolutionary Automatic Programming is used to refer to those systems that adopt evolutionary computation to automatically generate computer programs, and as such includes Genetic Programming (GP) and all its variants (e.g. Binary GP, AIM GP, developmental GP). We feel that the use of the term Evolutionary Automatic Programming, instead of GP with its various interpretations, is preferable when speaking about a GP system as it is not always clear as to the differences, if any, from the traditional tree-based Koza GP approach that operates directly upon the phenotypic trees [Koza, 1992]. This chapter describes the diverse array of approaches that have been adopted in this field of research and illustrate the success these techniques have enjoyed for the automatic generation of programs. We trace the evolution of this field from its humble beginnings evolving binary strings that specified a homegrown machine language, to the rich expressiveness of tree-based systems evolving Lisp S-expressions. We conclude by returning to string based systems involving evolution of C and machine code.

The field of evolutionary computation is based on the notion of biological evolution as described by Charles Darwin, that is, natural selection [Darwin, 1859] and incorporates Herbert Spencer's notion of survival of the fittest [Spencer, 1864]. Given a population of individuals where each individual represents a possible solution to a particular problem (e.g. a set of parameters), each individual is evaluated to determine its performance on the problem in question. The performance of an individual program on the problem is referred to as its fitness. Those individuals that have a superior performance on the task are assigned a better fitness value, and are most likely to survive by reproducing. It follows that these individuals are more likely to pass their genetic material to the next generation. The process of selecting individuals to reproduce, or in evolutionary computation to solve a problem, is referred to as natural selection. The idea being that over time reproduction and natural selection allow the evolution of increasingly fit individuals.

An evolutionary algorithm's population must be initialised before evolution can commence; typically this is conducted in a random fashion. For example, a random number generator is often used to set each bit value of every individual in a population of a simple Genetic Algorithm. In some cases it is desirable to seed the population with previous solutions, or attempt to confer a diversity of structure upon the first generation as commonly used in Genetic Programming, where variable length tree structures are adopted.

Each cycle of an evolutionary algorithm, otherwise referred to as a generation, is typically comprised of the phases *selection*, *genetic manipulation*, *testing* and *replacement*. During selection individuals are selected to be parents based on their fitness values, that is their ability, or lack of, at solving the target problem. Some genetic manipulation is then applied to these parents to generate children, followed by a testing phase that determines the fitness of each child solution. The replacement strategy determines how the children, parents and other members of the current population are used to create the next generation. An outline of an evolutionary algorithm can be seen below. New generations are continuously created while the stopping criteria are unfulfilled. These are taken as being either an adequate solution has been found, or a set number of generations have elapsed.

```
Initialise Population
WHILE termination criterion not satisfied
DO
    Evaluate fitness of each individual
    Select Parents
    Apply Genetic Operators
    Create new population
END DO
Report best-of-run individual
```

Evolutionary computation can be thought of as being comprised of a family of algorithms loosely based on the concept of survival of the fittest. The primary members of this family are the Genetic Algorithm (GA), Evolutionary Strategies (ES), Evolutionary Programming (EP), and Genetic Programming (GP).

The boundaries of distinction between each family member are becoming increasingly blurred. Traditionally, perhaps the most distinguishing factor of each of these algorithms was the representation that was adopted. In the case of GAs, these are traditionally fixed-length binary strings, in ES real-valued vectors, in EP individuals were originally finite state machines, and in GP, lisp S-expressions are traditionally the individuals of choice.

Current thinking suggests that, rather than concerning ourselves with distinguishing these members of the family, we should be able to combine the desirable properties from each of these approaches to suit the problem being tackled, by adopting an evolutionary algorithm [De Jong, 1999]. For example, using a linear GA-like genome to evolve lisp S-expressions.

As the main focus of this thesis is on Evolutionary Automatic Programming, we concentrate the following discussion on Genetic Programming and its variants. Before so doing we wish to place Genetic Programming within the historical context of Evolutionary Automatic Programming.

## 2.3 Origin of the Species

The concept of evolving executable computer programs dates back to Friedberg in 1958 [Friedberg, 1958], where "random changes" (analogous to what we now call mutations) and other "routine changes"[1] were made to binary string structures. These simple machine code programs were generated to solve simple arithmetic calculations, such as the addition of two numbers. Learning was implemented by using a credit assignment approach, whereby

[1]At any one time two possible instructions are kept on record for each instruction position in the program, one of which is active. The routine or deterministic changes involve swapping the instruction that is currently active for that position to its alternative instruction.

the performance of individual instructions was monitored. As noted in this work, no account was taken of the fact that there are interdependencies between instructions.

In [Friedberg et.al., 1959], two benchmark strategies for generating programs were compared for efficiency on their "learning machine" described in [Friedberg, 1958]. The first strategy made small changes by modifying a couple of instructions in the machine code. The second approach involved changing all the instructions within the code. It was found that the second, more aggressive, strategy was more efficient in terms of speed of obtaining correct programs. Results demonstrated that their original learning machine's performance was inferior to this second aggressive strategy, but on a par with the first. An attempt to improve the performance of their approach lead them to partitioning the problem into smaller sub-problems, leading to a performance that surpassed both of their benchmarks. It is worth noting that this approach adopted no notion of selection.

The next documented attempt to evolve computer programs directly was by Cramer in 1985 [Cramer, 1985], although [Fogel et.al., 1966] represented another milestone in the evolution of executable structures, through the evolution of finite state machines.

Cramer's goal was to devise a programming language that would be amenable to manipulation by a genetic algorithm and that should produce only well-formed programs (i.e. syntactically correct programs) [Holland, 1980]. Described in his work were two approaches: the JB language that adopted an integer list representation, and the TB language that used tree structures. It was found that JB did not satisfy Cramer's second goal of producing only well-formed programs. This was due to the semantic sensitivity of the position of each integer in the list that rendered the representation extremely

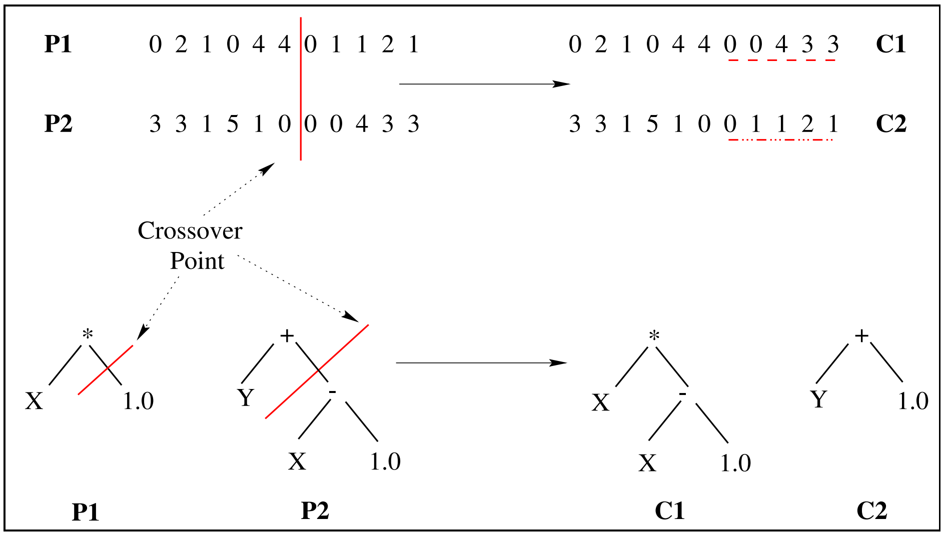


Figure 2.1: Examples of the crossover genetic operator at work. The top figure represents crossover operating on a linear string of integers, where P1 and P2 are the parents upon which a crossover point is selected. C1 and C2 are the two children produced as a result of the operator. The bottom figure represents crossover (sub-tree) as performed on tree structures.

brittle to change by mutation, and its seeming incompatibility with crossover due to its strong epistatic nature. An example of crossover operators in general can be seen in Figure 2.1. Taking the top example of crossover performed on a string of integers, in the case of the JB language, each integer would represent a program object (e.g. an operator, variable etc.) depending on the integers position. By performing crossover as illustrated it would be possible to produce a syntactically meaningless program, due to the dependencies on the order of previous objects and their semantics. Similarly for mutation, if one of these integers was mutated to another value, it is conceivable that this new value would result in the program being invalid.

TB, which exploited the tree-like structure of its programs, gave rise to the first sub-tree crossover operator for programs. In his concluding remarks, Cramer suggested that bringing the TB representation more in line with the linear nature of the JB representation would allow the exploitation of the power of the binary string genetic algorithm with it's simple genetic operators, or alternatively that it would be possible to extend the TB representation by adopting less standard genetic operators. No evidence exists within the literature of the field to suggest that Cramer followed up these ideas.

## 2.4 Tree-based Systems

Perhaps the most successful, and certainly the most widely adopted representation of computer programs for evolution is the Lisp S-expression, adopted by Koza in his seminal paper that was the forerunner for what is now referred to as Genetic Programming [Koza, 1989]. This paper stated that string representations had four fundamental problems:

1. They "do not provide the hierarchical structure central to the organization of computer programs (into programs and subroutines)";

2. They "do not provide any convenient way of representing arbitrary computational procedures or incorporating iteration or recursion";

3. They "do not facilitate programs modifying themselves and then executing themselves";

4. That "without dynamic variability, the initial selection of string length limits in advance the number of internal states of the system and the computational complexity of what the system can learn".

Given what had been achieved with linear representations up to this point these statements were correct, but later we will see advances that have allowed linear representations to overcome the problems identified by Koza.

### 2.4.1 Genetic Programming

Since its inception, Genetic Programming (GP) has enjoyed much popularity and to date (March 2001) has resulted in the publication of twelve authored books, three edited books, nine conference proceedings, over 1,500 papers, its own journal *Genetic Programming and Evolvable Machines* published by Kluwer Academic Publishers, and thirty-seven completed PhD theses, according to its official website [Koza, 2001].

For a thorough description of GP one should refer to Koza's three books, Genetic Programming 1, 2 and 3 [Koza, 1992][Koza, 1994][Koza et.al., 1999], and the first GP text book by Banzhaf et. al. [Banzhaf et.al., 1998].

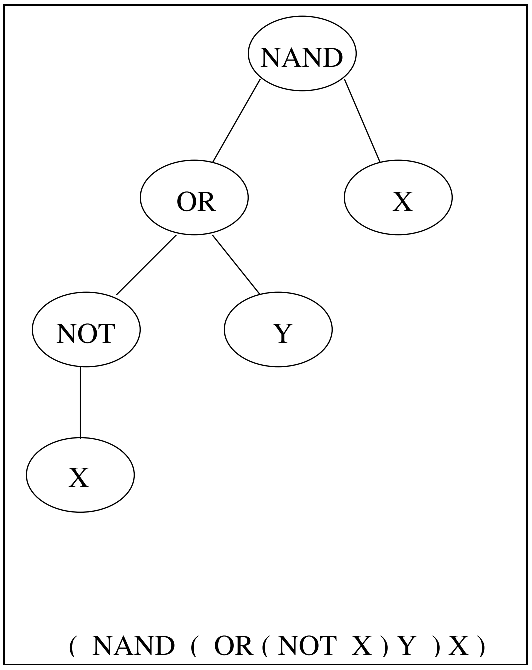


Figure 2.2: An example parse tree of Genetic Programming and the corresponding Lisp S-expression.

The essence of GP is to evolve more complex structures that represent computer programs. The standard version of GP is comprised of a population of Lisp S-expressions represented in the form of parse trees, see Figure 2.2. The parse tree representation facilitates the genetic manipulation of the evolving programs, by using a crossover operator applied to sub-trees, as opposed to solely relying on mutation. In order to evolve a computer program by means of GP, one must specify the primitives of the system, referred to by Koza as the function and terminal sets, before designing a fitness function appropriate to the problem domain. Elements of the terminal set have an arity of zero, that is they return a numerical or boolean value without taking input values themselves.

An example terminal set, denoted by $T$, could be as follows:

$$T = \{x, y, z, 1.0\}$$

Note that $T$ contains three variables x, y, and z and a constant 1.0.

The function set, usually denoted as $F$, is comprised of entities such as operators and functions that usually have an arity greater than zero, that is they can take input values. A function set can contain boolean operators, arithmetic operators, subroutines etc. For example:

$$F = \{NAND, +, -, *, /, moveForward(), turnLeft(), turnRight()\}$$

When choosing primitives one must ensure that the property of *sufficiency* holds, that is, the set of functions and terminals must be able to represent a solution to the problem. One must also ensure that the property of *closure* holds for the function set, that is, each element of the function set must be able to handle all possible input values it may receive, given the total set of primitives available to it and the return values of all other functions. While not essential, it is wise to ensure one's function set is parsimonious as too large a function set can slow down evolutionary search by needlessly creating a very large search space, as has been demonstrated with GPPS [Koza et.al., 1999].

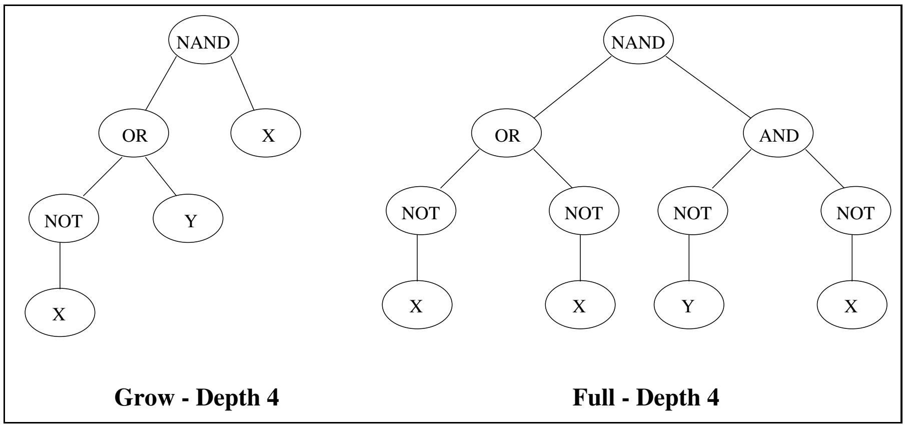


Figure 2.3: The GP individual generation techniques Grow and Full compared for a tree depth of 4.

After selecting the primitives one must specify standard parameters to evolutionary search such as probabilities for genetic operators, population sizes, and selection and replacement mechanisms. In addition, one must specify the maximum tree depths and the type of initialisation method used. Koza outlined the *ramped half-and-half* method for the initialisation of a GP population. The aim of this technique is to facilitate structural diversity in the population, by creating individuals with a uniform distribution of varieties of tree depths. The population is divided equally amongst individuals of a range of tree depths, and then for the number of individuals assigned to a particular depth, half are created using a technique called *grow* and the other half using the *full* technique. With the full technique every tree branch will grow to the full depth allowed resulting in a uniform tree, while the grow technique results in individuals whose branches can be of varying depths up to the maximum depth specified for that group; see Figure 2.3 for examples.

An outline of the standard GP algorithm can be seen in Section 2.2. This description represents the original version of GP, as outlined by Koza; since then there have been many additions and extensions to tree-based GP, such as the use of ADF's [Koza, 1994], Indexed Memory [Teller, 1994], Strongly typed GP [Montana, 1995], etc.

We will now discuss some of the extensions to tree-based GP, focusing on those approaches whose concepts play a part in the Grammatical Evolution system that is the focus of this thesis.

### 2.4.2 Grammar based Genetic Programming

The use of grammars with GP has enjoyed much popularity in recent years. Various approaches have been made using different classes of grammars. Before highlighting the systems that combined grammars with GP we will first give an introduction to grammars, and their standard notation. [Chomsky, 1956] identified a number of different classes of languages and their grammars which he placed in a hierarchy. In increasing complexity these languages are the regular (type 3), context-free (type 2), context-sensitive (type 1), and arbitrary (type 0), where each language is a subclass of the next. In particular, those grammars that are said to be either context-sensitive or context-free are the ones most commonly adopted in GP. These grammars can be represented in Backus Naur Form [Naur, 1963]. The grammars used in GP are typically used in the generative sense to construct a program.

#### Backus Naur Form

Backus Naur Form (BNF) is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of `terminals`, which are items that can appear in the language, e.g. $+$, `-` etc.

and `non-terminals`, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple $\{N, T, P, S\}$, where $N$ is the set of non-terminals, $T$ the set of terminals, $P$ a set of production rules that maps the elements of $N$ to $T$, and $S$ is a start symbol which is a member of $N$. When there are a number of productions that can be applied to one element of $N$ the choice is delimited with the '|' symbol. For example,

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, +, -, /, *, X, 1.0, (,)\}$$

$$S =< expr >$$

And $P$ can be represented as:

```
<expr> ::= <expr> <op> <expr>          (a)
         | ( <expr> <op> <expr> )      (b)
         | <pre-op> ( <expr> )         (c)
         | <var>                       (d)


<op> ::= +                             (e)
       | -                             (f)
       | /                             (g)
       | *                             (h)


<pre-op> ::= sin                       (i)
           | cos                       (j)


<var> ::= X                            (k)
        | 1.0                          (l)
```

Unlike the approach in [Koza, 1992], there is no distinction made at this stage between functions (operators in this sense) and terminals (variables in this example); however, this distinction is more of an implementation detail than a design issue. Whigham [Whigham, 1996a] also noted the possible confusion with terminology and used the terms **GPFunctions** and **GPTerminals** for clarity.

In many of the examples to follow, individuals in a population are represented as derivation steps. A derivation step is simply the application of a production rule, $P$, to an element of the non-terminal set, $N$. This transformation is denoted by the $\Longrightarrow$ symbol, with the rule being applied written above it.

$$<expr><op><expr> \overset{Rule\ (d):\ <expr>::=<var>}{\Longrightarrow} <var><op><expr>$$

Above is an example of a derivation step in which we see the application of the rule $Rule\ (d) : <expr> ::=<var>$ to the leftmost non-terminal $<expr>$. This results in $<expr>$ being replaced with $<var>$.

An example of a derivation tree that would comprise an individual of a population in some of the following cases can be seen in Figure 2.4.

This derivation tree can easily be converted into the more familiar parse tree format adopted by tree-based GP, see Figure 2.5. It is more difficult, however, to convert a parse tree into an equivalent derivation tree, as the process is usually non-deterministic.

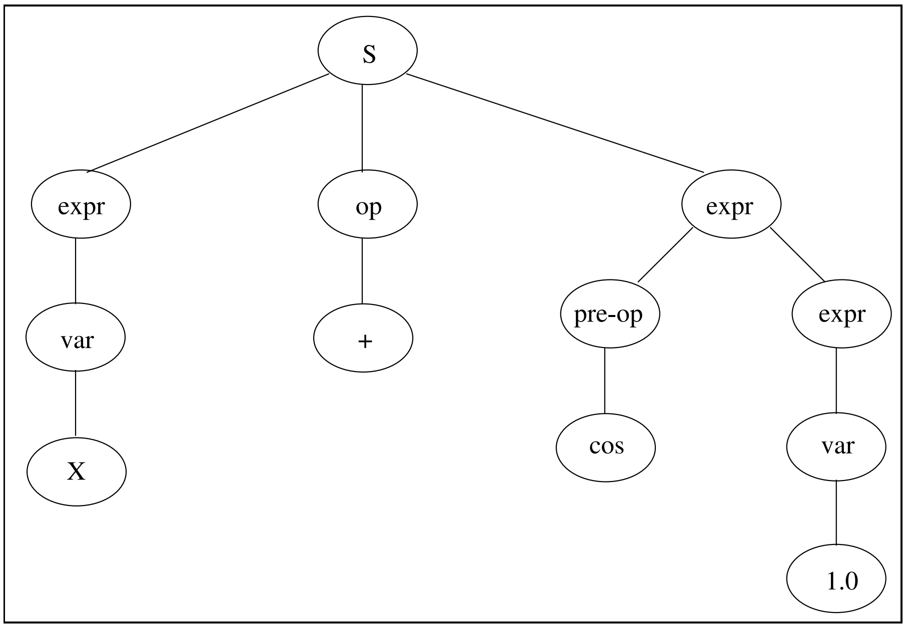


Figure 2.4: An example of a derivation tree for a grammar-based GP individual.

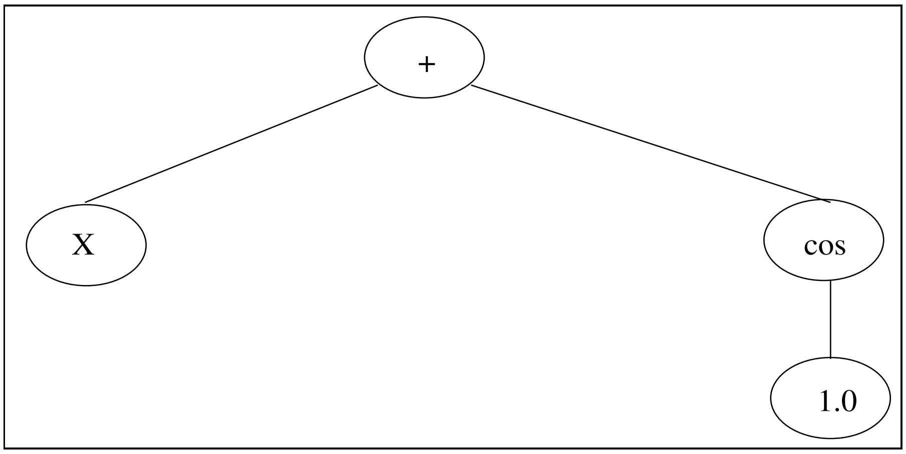


Figure 2.5: The example derivation tree of Figure 2.4 converted into its equivalent parse tree.

### Cellular Encoding

Gruau describes an approach that enables the representation of neural networks in the form of trees, which made it possible to manipulate them by GP [Gruau, 1994]. The approach represents neural networks in the form of cellular encoding, which in turn is represented in the form of a graph grammar. This is used to generate the trees of the GP population.

A graph grammar represents a set of rules for re-writing the topology of a graph. This is an embryonic approach to the generation of the neural networks, where each cell of the neural network corresponds to a node of the graph being constructed by the grammar. Starting from an ancestor cell, rules of the graph grammar are applied to cells, resulting in the division of the ancestor cell into further cells with properties that may differ from those of the original cell. The final result is a complete neural network where the evolutionary process alone determines its size and functionality. Koza et.al. have adopted a similar embryonic approach to the evolution of analog circuits [Koza et.al., 1999].

### Auto-parallelisation with GP

One interesting application/variation of tree-based GP has been its use in the area of auto-parellisation of serial code. A transformational approach is adopted in a manner not dissimilar to that of Gruau [Gruau, 1994]. The essence is to transform a functionally correct serial program into a functionally equivalent parallel version. Each individual in the population is a set of transformations that are used to carry out the process of parallelisation, and so the best set of transformations is evolved.

Note, however, that this approach does not use grammars. There have been numerous publications on this subject [Ryan & Walsh, 1997] [Ryan & Ivan, 1999] [Ryan, 1999].

### Bias in GP

The work of Whigham represents an application of grammars to tree-based GP as a means to incorporate declarative and learnt bias [Whigham, 1996b] [Whigham, 1995a]. Declarative bias is the representation of knowledge separate from the learning system, and is provided by the initial grammar given to the system. Over the evolutionary search it is possible to incorporate changes into the grammar and this is what constitutes the learnt bias. Bias, in the form of the context-free grammar, is used in the creation of the initial population to narrow the possible representations that the system may adopt. The grammar is also used during the application of the genetic operators to ensure the syntactic correctness of individuals.

### Combining GP and ILP

Wong and Leung describe an approach that employs logic grammars with tree-based GP, referred to as LOGENPRO (LOgic grammars based GENetic PROgramming system) [Wong & Leung, 1994] [Wong, 1995] [Wong & Leung, 1997]. Billed as being a framework to combine GP and Inductive Logic Programming (ILP), LOGENPRO has enjoyed much success in its application to the area of data mining [Wong & Leung, 2000]. Derivation trees, as in Figure 2.4, are the representation of choice for individuals in the population. Logic grammars differ from standard context-free grammars in that both the terminals as well as the non-terminals can include arguments. Arguments in the grammar can be used to force context-dependency, thus the logic grammar is a member of the class of context-sensitive grammars. The arguments can also be used to generate tree structures in the course of parsing which in turn can be used to infer semantics of the program.

### Genetic Programming Kernel

Horner introduced an implementation of a grammar-based GP system, called Genetic Programming Kernel (GPK) [Horner, 1996] [Geyer-Schulz, 1997], similar to Whigham's approach, by virtue of its use of context free grammars. Again, individuals in the population are derivation trees constructed according to a BNF definition.

The GPK has been criticised [Paterson & Livesey, 1997] for the difficulty associated with creating the first generation - considerable effort must be put into ensuring that all the trees represent valid sequences, and that none grow without bounds.

## 2.5 String based GP

In parallel to developments with tree-based GP, major advances have been made with the use of string based Evolutionary Automatic Programming systems. One of the major reasons for the continuation of research in this area is the potential benefits that can arise from a separation of the genotype and phenotype that does not exist in tree-based systems. In GP, evolution is generally conducted upon the actual programs that are the product of the process (the phenotype). If we were to allow evolution to be conducted upon another representation of the program (the genotype) it must be mapped onto a functional program (phenotype). A full discussion on this is deferred until Chapter 3.

Many of the advances made in linear systems have been accompanied by the adoption of grammars for much the same reasons as those in tree-based GP systems, that is, to ensure the syntactic correctness of evolving programs. As Grammatical Evolution, the focus of this thesis, adopts grammars we will focus in particular on those linear systems that use grammars to some degree.

### 2.5.1 BGP

Banzhaf [Banzhaf, 1994] and later [Keller & Banzhaf, 1996] described a linear GP system that employed a genotype-phenotype mapping from a linear genotype into a linear phenotype, using a context free programming language. The system, referred to as Binary Genetic Programming (BGP), adopts a context-free grammar for a repair procedure during the mapping from the linear genome to the output language.

The genotype in this case is a binary string that contains *codons* of a predetermined number of bits, each of which represents a symbol of the output language. The mapping of codon to symbol is explicitly determined at the outset. In a manner similar to the biological code, they adopt a redundant genetic code, that is, many different codons can represent the same language symbol. An example of such a mapping can be seen in Figure 2.6.

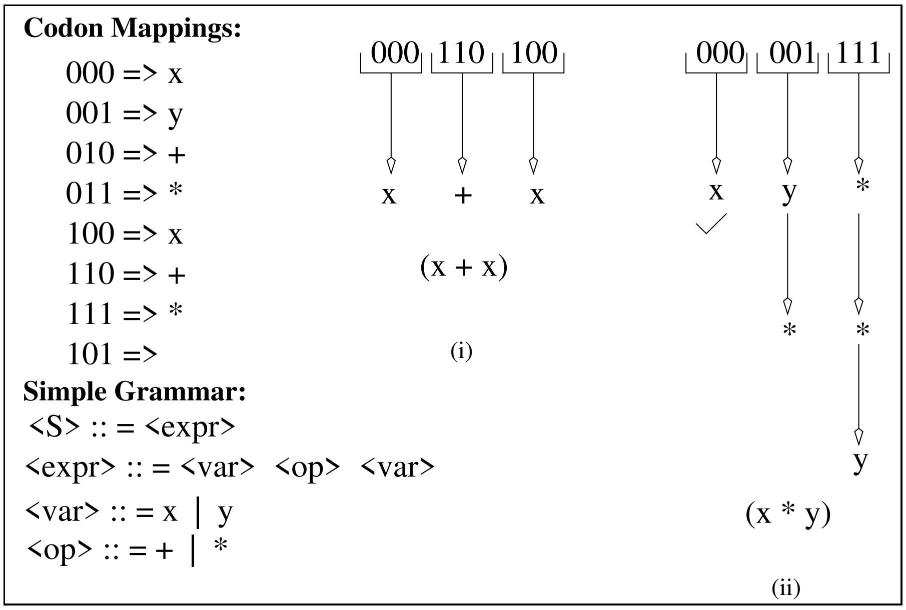


Figure 2.6: Example 3 bit codon codes with their corresponding symbols for Binary Genetic Programming. Note redundancy in the mapping. (i) Illustrates a straightforward mapping from codons to a valid expression $x + x$. (ii) Demonstrates the repair mechanism, where the second symbol $y$ is illegal in this position and must be replaced with a legal symbol according to a grammar. In this case valid symbols for this position could be $*$ or $+$. $y$ is replaced with the symbol whose codon has the closest hamming distance (bit flips), i.e. $*$. The third original symbol, $*$, is now illegal and so the repair mechanism is employed resulting in it being replaced with $y$.

Their experiments suggest benefits, in terms of improvement in performance (higher fitness), in adopting a genotype-phenotype mapping over standard genetic programming and that further analysis of the genetic codes was required. It was not clear as to what, if any, effect the redundancy in the genetic code was playing in the success of this system. Later [Keller & Banzhaf, 1999] an investigation was conducted in which it was possible to evolve the genetic code itself. It was found that the code tended towards redundancy for certain symbols, that is, evolution was able to learn the significance of those symbols for the problem being addressed. The ability to evolve the genetic code during the evolutionary search has the potential to enhance the ability to solve dynamic problems, as changes in the genetic code could enable the search process to keep up with the changing topology of the search landscape. The authors suggest that the ability to evolve the genetic code could be of particular use in data mining applications where the functional relation between variables are unknown, and as such could enhance the learning of significant functional relations.

### 2.5.2 Machine Code Genetic Programming

A machine code, linear GP system originally called the Compiling Genetic Programming system, now AIM-GP(Automatic Induction of Machine Code for Genetic Programming), has been developed that is up to 1000 times faster than a standard GP system [Nordin, 1994] [Nordin & Banzhaf, 1995] [Francone et.al., 1996] [Nordin, 1997] [Nordin, 1998] [Nordin et.al., 1999]. A commercial version of AIM-GP has been recently released under the name of Discipulus$^{TM}$ [RML].

AIM-GP adopts a linear binary string genome to maximise efficiency gains, as it means the individuals (machine code programs) can be directly evaluated on the hardware architecture of the machine without having to undergo either a mapping or interpretation stage. Like a standard tree-based GP system, evolution acts directly upon the program with no genotype-phenotype distinction. Results on various classification problems have shown that CGP's performance is comparable to other Machine Learning paradigms, such as classifier systems, and multilayer feedforward neural networks.

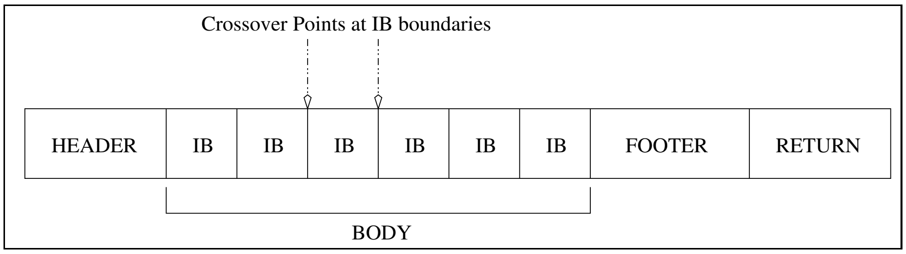


Figure 2.7: An AIM-GP individual is a machine code function comprised of a header, a body containing one or more 32-bit instruction blocks (IB's), a footer and a return block.

Each individual in AIM-GP is a machine code function comprised of four parts, a header and footer, the main function body itself containing many instruction blocks, and a return statement (see Figure 2.7). Evolution is directed towards the 32 bit instruction blocks of the function's body, each of which can contains one or more instructions of 8, 16, 24 or 32 bits with a combined size of 32 bits. The genetic operator of crossover is restricted to the boundaries of the instruction blocks, and mutation acts by either randomly generating a new instruction block, or by randomly changing an operand or operator of one of the instructions. Two variants of crossover are adopted, a standard two-point, and a homologous crossover that swaps instruction blocks located at the same locus [Francone et.al., 1999].

### 2.5.3 Genetic Algorithm for Deriving Software

Paterson & Livesley described a Genetic Algorithm (GA) approach to the automatic generation of programs called the Genetic Algorithm for Deriving Software (GADS) [Paterson & Livesey, 1996] [Paterson & Livesey, 1997]. Like Binary Genetic Programming, described in Section 2.5.1, fixed-length, binary, linear genomes are employed with a genotype-phenotype mapping process to generate programs, but BNF Grammars are utilised in GADS as the output language specification. The BNF of GADS is extended to include a default symbol for each non-terminal of the grammar. The default symbols are used if the situation arises where all the genes of an individual have been read during the mapping process and the mapping is incomplete, that is, elements of the non-terminal set still appear in the expression being mapped. The default rules are then applied in order to complete the mapping process.

During the mapping from genotype to phenotype, a parse tree is generated, by initialising the root node to the start symbol of the grammar. Starting from the left-most gene, the integer value of each gene is read, and if the value corresponds to a suitable production rule that rule is applied to the current non-terminal. If the gene value does not correspond to an appropriate rule the gene is skipped and another read. The mapping process terminates upon reaching the end of the individual's chromosome, at which point any remaining non-terminals are replaced with their default symbols. Paterson & Livesley concede that a weakness in this approach lies in the fact that given large grammars, the likelihood of any one particular production rule being selected is diminished. In order to improve the chances of specific rules being selected, rule weighting, achieved by the duplication of some productions in the grammar to promote their use, was adopted. Paterson also employed the strategies of increasing population sizes and increasing individual lengths as a measure to counteract this problem. Another potential weakness of this approach is the manner in which genes are skipped if unsuitable for the current non-terminal. This often results in a proliferation of introns that can choke effective evolutionary search, and can also leave the system open to a reliance on the user-defined default terminals. As one does not always know what would be the ideal terminal in every situation for each non-terminal in the grammar, it would be a more desirable approach to let evolution decide what terminals are adopted.

### 2.5.4 CFG/GP

Freeman describes an approach called Context Free Grammars GP (CFG/GP) that is very similar to GADS [Freeman, 1998]. Again, fixed length linear genomes are employed where genes are integer values. There is a subtle difference during the mapping process that distinguishes it from GADS; instead of attempting to apply rules specified by each gene to the next non-terminal (left-most), each gene is read and can be applied to any suitable non-terminal in the partially mapped parse tree. Given the similarity to GADS, this system suffers from similar drawbacks in terms of scalability of the grammar and the likelihood of specifying any one rule, the proliferation of introns although not necessarily to the same extent given the subtle difference in approach during the mapping from genotype to phenotype, and the use of pre-determined default rules.

## 2.6 Conclusions

The above is by no means an exhaustive representation of all the approaches adopted in Evolutionary Automatic Programming, it is rather a sample of those systems of particular importance and relevance to the subject of this thesis, that is, automatic programming in an arbitrary language. Without some of the developments outlined above it would not have been possible to arrive at the Grammatical Evolution system; we owe much inspiration to those whose work has gone before us. Before detailing the Grammatical Evolution system in Chapter 4, we will discuss what we consider to be the fundamental characteristics of an Evolutionary Automatic Programming system, by paying particular attention to lessons that can be learned from Molecular Biology.

# Chapter 3

# Lessons from Molecular Biology

## 3.1 Introduction

We will now discuss some fundamental principles from Molecular Biology that can be exploited by an Evolutionary Automatic Programming system. In particular we focus on those aspects that have provided inspiration, in conjunction with previous work (see Chapter 2) for the Grammatical Evolution system that is the focus of this thesis. We will see later in Chapter 4 how some of these principles have been adopted successfully by the Grammatical Evolution system.

In recent years there has been an increase in the body of research conducted in the area of gene expression in evolutionary computation. Notable areas include genotype-phenotype distinction [Ryan et.al., 1998] [Banzhaf, 1994] [Gruau, 1994], genetic-code evolution [Keller & Banzhaf, 1999], distributed fitness evaluation [Kargupta & Sarkar, 1999], diploidy [Ryan & Collins, 1998] [Collins & Ryan, 1999] [Goldberg, 1987] [Hollstein, 1971] [Ng & Wong, 1995] [Ošmera et.al., 1997], role of introns [Kargupta, 1997], degenerate genetic codes and neutral mutations [O'Neill & Ryan, 1999] [Barreau, 2000] [Barnett, 1997] [Banzhaf, 1994] [Vassilev & Miller, 2000].

In the context of Evolutionary Automatic Programming, a distinction of genotype and phenotype is perhaps the most powerful area that can be exploited. We outline a list of desirable features that such a system could possess, and describe how these can be gained from such a distinction in Section 3.4. Arising from a genotype-phenotype distinction, issues such as the role of the genetic code and gene expression play increasing importance. Before moving on to discuss the potential benefits that can be obtained from a genotype-phenotype distinction we will firstly discuss the genetic code and gene expression models of biological systems as determined through research in the field of Molecular Biology. This will be followed by a discussion on the neutral theory of evolution and the possible benefits this phenomenon can bring to evolutionary computation.

## 3.2 Genetic Codes & Gene Expression Models

Molecular Biology, a relatively young scientific field, has yielded many insights into the innermost workings of the fundamental units of complex organisms, cells. Discoveries include the structure of the hereditary material DNA, the discovery of proteins and their structural units, the elucidation of the genetic code, and the complex signalling networks that exist within and between cells. Of utmost importance is what has become known as *the central dogma of Molecular Biology*. This tenet describes the processes of transcription and translation, i.e. that DNA is transcribed to mRNA, which in turn is translated into protein.

Proteins are responsible for the generation of phenotypic traits, such as eye colour, and height.

DNA contains the genetic code that specifies the constituent units of proteins, called amino acids. Each amino acid, and their sequence within a protein, is represented by a particular sequence of the molecules that make up the genetic code. Specifically, groups of three molecules, called bases, code for a single amino acid. A group of three bases is referred to as a codon. A base is one of the molecules Adenine (A), Thymine (T), Guanine (G), and Cytosine (C) in DNA, and Uracil (U) instead of Thymine in RNA.

Table 3.1 outlines the complete human genetic code for both DNA and RNA. Proteins are generated during the process of translation that literally translates the base triplets into their corresponding amino acids. There are 20 naturally occurring amino acids and 64 triplet sequences, called codons. Three of these codons are used to specify the termination of translation and do not generally specify amino acids. We are left with 61 codons for 20 amino acids. As a result the genetic code is degenerate, that is, there is a many-to-one mapping such that an amino acid can be specified by many different codons. This occurs through a phenomenon known as the *wobble hypothesis* [Crick, 1966], and it means that a mutation at the third position in a codon does not always result in the code for a different amino acid.

Such mutations are referred to as *silent* or *neutral* mutations, and these have possible implications for evolutionary search and dynamics [Kimura, 1983].

There has been some work recently on the evolutionary consequences of degeneracy[1] (referred to as redundancy in previous work in the evo-

[1]We recommend the adoption of the term genetic code degeneracy as this is the terminology used in Molecular Biology, and is a more correct description of the phenomenon. Redundancy is usually used in the context of genetic material surplus to requirement in the field of evolutionary computation e.g. introns. As such, we feel it is important to draw

| | **U** | **C** | **A** | **G** | |
|---|---|---|---|---|---|
| **U** | UUU - Phe | UCU - Ser | UAU - Tyr | UGU - Cys | **U** |
| | UUC - Phe | UCC - Ser | UAC - Tyr | UGC - Cys | **C** |
| | UUA - Leu | UCA - Ser | UAA - Stop | UGA - Stop | **A** |
| | UUG - Leu | UCG - Ser | UAG - Stop | UGG - Trp | **G** |
| **C** | CUU - Leu | CCU - Pro | CAU - His | CGU - Arg | **U** |
| | CUC - Leu | CCC - Pro | CAC - His | CGC - Arg | **C** |
| | CUA - Leu | CCA - Pro | CAA - Gln | CGA - Arg | **A** |
| | CUG - Leu | CCG - Pro | CAG - Gln | CGG - Arg | **G** |
| **A** | AUU - Ile | ACU - Thr | AAU - Asn | AGU - Ser | **U** |
| | AUC - Ile | ACC - Thr | AAC - Asn | AGC - Ser | **C** |
| | AUA - Ile | ACA - Thr | AAA - Lys | AGA - Arg | **A** |
| | AUG - Met | ACG - Thr | AAG - Lys | AGG - Arg | **G** |
| **G** | GUU - Val | GCU - Ala | GAU - Asp | GGU - Gly | **U** |
| | GUC - Val | GCC - Ala | GAC - Asp | GGC - Gly | **C** |
| | GUA - Val | GCA - Ala | GAA - Glu | GGA - Gly | **A** |
| | GUG - Val | GCG - Ala | GAG - Glu | GGG - Gly | **G** |

Table 3.1: The Genetic Code of biological organisms. The full names for the amino acids are given in Table 3.2. Notice the redundancy that occurs for most of the 20 naturally occurring amino acids. In general, this code can be considered universal across most species, although variations do arise.

| Code | Name | Code | Name |
|---|---|---|---|
| **Phe** | Phenylalanine | **Leu** | Leucine |
| **Tyr** | Tyrosine | **Cys** | Cysteine |
| **Trp** | Tryptophan | **Pro** | Proline |
| **His** | Histidine | **Gln** | Glutamine |
| **Arg** | Arginine | **Ile** | Isoleucine |
| **Met** | Methionine | **Thr** | Threonine |
| **Asn** | Asparagine | **Lys** | Lysine |
| **Ser** | Serine | **Val** | Valine |
| **Ala** | Alanine | **Asp** | Aspartic Acid |
| **Glu** | Glutamic Acid | **Gly** | Glycine |

Table 3.2: The three letter amino acid codes as in Tables 3.1 with their corresponding full amino acid names.

lutionary computation community) in natural and artificial genetic codes [Barreau, 2000] [O'Neill & Ryan, 1999] [Vassilev & Miller, 2000]. The major consequences of neutral evolution and degenerate genetic codes for evolutionary computation are discussed in Section 3.3, which follows.

Ongoing areas of investigation are the regulatory processes that exist to control the expression of particular proteins from specific genes along the DNA in different organisms. The discovery of special regions along the DNA yielded the formation of the operon model [Jacob & Monod, 1961]. These regions known as repressors, promoters, and operators are involved in the regulation of the expression of specific genes, e.g. the well-known prokaryotic lac operon that contains three genes associated with the metabolism of lactose. An overview of the operon model can be seen in Figure 3.1.

What is crucial to note when dealing with gene expression, is that it occurs within a cellular environment with complex feedback loops controlling the further expression of genes. While all cells in an organism have identical genetic material, each cell has the ability to specialise in terms of functionality and structure, depending on what genes are expressed at certain stages of cell development. In [Kennedy, 1998] a novel artificial model of a biological cell is presented in which each cell has a genome and metabolism. Pairs of these genomes and metabolisms co-evolve using a genetic algorithm. A simple genomic language is introduced that follows Jacob and Monod's operon model [Jacob & Monod, 1961], such that genes can be placed at any loci on the genome. Thus a model is generated in which the genome specifies proteins such as enzymes that can catalyse metabolic reactions, and the chemicals of metabolism have the ability to regulate genes and allow expression of proteins from the genome. The model, partly Lamarckian due to

the distinction between these two subtly different but related phenomenona.

the manner in which the metabolism evolves, is found to be more efficient at solving the boot-strapping problem due to the co-evolution of the metabolism with the genome. The bootstrapping problem involves finding genomes and metabolisms that can work together in a synergistic fashion to result in stable cells for a target environment.

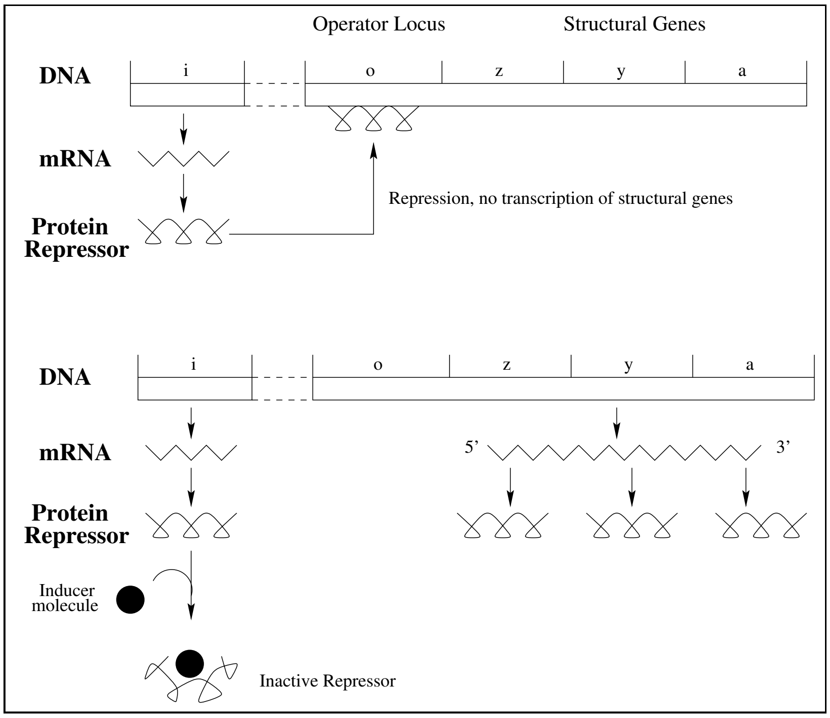


Figure 3.1: A diagram illustrating a modified form of the operon model as described by Jacob and Monod ($i$ was thought to be an mRNA rather than a protein). The $i$ gene encodes a repressor that binds tightly to the operator $o$ locus, thereby preventing transcription of the mRNA from the $z$, $y$, and $a$ structural genes. When inducer is present, it combines with repressor, changing its structure so it can no longer bind to the operator locus, thus allowing translation of the three structural genes. The inducer also has the ability to remove repressor already complexed with the $o$ locus.

## 3.3 Neutral Theory of Evolution

A recent upsurgence of interest in the role of selectively neutral evolution in evolutionary dynamics has brought about the concepts of *neutral mutations* and *neutral networks*. A neutral mutation is a mutation event, typically a point mutation that has no effect upon the phenotypic fitness of an individual in a population. The term neutral network has been coined to describe a group of individuals that are connected by neutral mutations.

Kimura was perhaps the first to recognise the potential neutral mutations have to effect evolution [Kimura, 1983]. He devised the neutral theory of molecular evolution, which stated that most of evolution occurs via mutations that are neutral with respect to selection, and thus evolution is occurring via the resulting random genetic drift. An interesting observation arising from this is that given a continuous variation of genetic material, one would have a mechanism to explain the occurrence of genetic diversity within natural populations. In the field of Genetic Programming, Banzhaf took neutral theory into account when employing a genotype to phenotype mapping, in order to generate genetic diversity within his artificial populations allowing his algorithm to escape local optima [Banzhaf, 1994]. The ability for neutral mutations to occur in Banzhaf's algorithm arose due to the use of a hand coded degenerate genetic code.

EA practitioners adopt the notion of *fitness landscapes*, as do biologists. These fitness landscapes allow us to visualise evolutionary dynamics so one could say, for example, that selection acts as a force driving a population up a fitness peak, while mutations serve to disperse the population, counteracting to some extent the selective force. One problem for the evolutionary computation community is the problem of a population becoming trapped on a local optimum due to the overpowering strength of the selective force.

Given that in biological populations this does not appear to be the case, this suggests that a piece(s) of the puzzle is missing and, as such, our models of evolutionary dynamics over these fitness landscapes would appear to be incomplete. For example, in [Cructhfield & van Nimwegen,1999b] evidence has been presented to suggest that a population undergoing evolution on a fitness landscape with neutrality tends to concentrate itself at highly connected regions of the neutral networks. This allows mutation to occur, changing the genotypic sequence, while ensuring preservation of the functionality.

Work by Kimura, Eigen and others on RNA evolution suggests that the existence of neutral networks gives a different perspective on evolutionary dynamics over a fitness landscape [Eigen et.al., 1989] [Huynen, 1995] [Huynen et.al., 1996] [Cructhfield & van Nimwegen, 1999a] [Cructhfield & van Nimwegen,1999b] [Reidys, 1995] [Reidys et.al, 1997] [Reidys et.al., 1998]. Rather than the traditional view of a population performing hill climbing, we now have the scenario where a population could also be drifting along these neutral networks with the occasional jump between adjacent networks. Of particular significance is when these networks are said to *percolate* a landscape, that is to say, they are arbitrarily close to every other neutral network, then given time almost any possible fitness can be attained by the population. Artificial system simulations of the evolution of molecular species [Engelhardt & Newman, 1998] [Engelhardt, 1998] state the claim that the maximum fitness attainable during evolution on a fitness landscape with neutrality increases with increasing degrees of neutrality, and is also directly related to the fitness of the most fit percolating network. This finding could prove to be beneficial to those problems of a dynamic nature or where an ideal fitness is difficult to define or unknown.

In the next section we will present potential advantages to be gained by incorporating principles from Molecular Biology, discussed up to this point, into our algorithms.

## 3.4 Desirable Features

We will now outline some of the desirable features that an Evolutionary Automatic Programming system might gain with the incorporation of the previously discussed principles from Molecular Biology.

1. Generalised encoding that can represent a variety of structures.
2. Efficiency gains for evolutionary search.
3. Maintenance of genetic diversity within an evolving population.
4. Preservation of functionality while allowing continuation of search.
5. Re-use of genetic material.
6. A compression of representation.
7. Alternative implementation of functions.
8. Positional Independence.

The first feature, a generalised encoding that can represent a variety of structures, is the main advantage for Evolutionary Automatic Programming. This could be achieved with an abstraction of our genotype from the output program, thus allowing a mapping process that can be utilised to generate code in an arbitrary language.

In order to achieve efficiency gains for the evolutionary search, the second feature above, the adoption of a degenerate genetic code could prove useful. If the output program is not directly represented within the genotype, as would be the case if our first feature was implemented, it would be possible to adopt a degenerate code. This can lead to efficiency gains through the use of neutral mutations and their corresponding neutral networks, see [Barnett, 1997] [Engelhardt & Newman, 1998]. A degenerate code exists when many different codes are in existence that corresponds to the same phenotypic symbol. Given findings in [Engelhardt & Newman, 1998] that given a higher degree of neutrality greater fitness's can be attainable, it would be interesting to incorporate a genetic code with a tuneable degree of neutrality.

As found in [Banzhaf, 1994] and suggested by [Kimura, 1983] a degenerate code can also facilitate genetic diversity, the third of the features we have outlined above. Genetic diversity could be maintained during runs through the distinction of genotype and phenotype. Maintenance of genetic diversity is of particular importance in the field of evolutionary computation to overcome the problem of premature convergence. A population is said to have converged if it has become stuck on a local optimum with no means of freeing itself. There can be no further evolution in this case and unless the local optimum happens to correspond to the global optimum the evolutionary search has failed. For a discussion on strategies to prevent premature convergence refer to [Ryan, 1996].

Evolutionary search can be facilitated by enabling the preservation of functionality (fourth feature), thus allowing individuals to survive due to their phenotypic fitness while still allowing variations in their genotype. This effect can be facilitated when a genotype-phenotype distinction exists with the presence of a many-to-one mapping from genotype to phenotype. One example of this occurring is when evolutionary search is moving along a neutral network as discussed earlier. The mapping process can also either incorporate some form of a repair mechanism or inherently have the ability to ensure that no matter what happens to the genotype a valid phenotype will always be generated. Refer to [Yu & Bentley, 1998] for a discussion on generating legal phenotypes from a genotype.

The fifth feature, a re-use of genetic material, can be achieved with ease when a genotype-phenotype distinction is present. The same genetic material can be used during any one mapping from genotype to phenotype. In Chapter 2, examples of linear GP systems were given where upon reaching the end of the genotype during mapping and an incompletely mapped individual remains, pre-determined default symbols were substituted where gaps existed. Having the possibility of gene expression at hand it would be possible to re-use some or all of the genetic material in order to attempt a completion of the mapping process. We subscribe to the notion that it is preferable to let the evolutionary process decide what to do in these cases, rather than settling for pre-determined values.

Given the existence of the fifth feature, it would be possible to attain the sixth feature of a compression of representation. That is large structures can be represented with a relatively small genotype.

Through the implementation of the operon model into a system containing a genotype-phenotype distinction our seventh feature could exist. Looking to Molecular Biology, gene expression mechanisms could be adopted to allow an equivalent or improved version of GP's automatic function definition [Koza, 1994]. In effect, alternative transcription algorithms could be incorporated.

The eight feature, positional independence, amounts to the position (locus) of a gene on the chromosome bearing no effect on its functionality. For example, this useful phenomenon could be achieved using the operon model by allowing metabolites to recognise a specific promoter region on the chromosome, thus switching on or off that particular gene (or set of genes) expression. This could also be achieved through the use of novel gene expression mechanisms and genetic codings. In terms of Evolutionary Automatic Programming, positional independence could give rise to more productive crossover events due to the separation of locus and function.

## 3.5 Conclusion

We have identified some of the benefits that could be made available to an Evolutionary Automatic Programming system that adopts the principles from Molecular Biology outlined in this chapter. Most of the benefits that can arise are due to a genotype-phenotype distinction. In the following chapter we will show how many of the advantages outlined above are incorporated into the Grammatical Evolution system. In subsequent chapters we will report on an analysis of some of these features as they appear in Grammatical Evolution and demonstrate how they contribute to its success.

# Chapter 4

# Grammatical Evolution

## 4.1 Introduction

We present Grammatical Evolution (GE), an evolutionary algorithm that can evolve complete programs in an arbitrary language using a variable-length binary string. The binary genome determines which production rules in a Backus Naur Form (BNF) grammar definition are used in a genotype-to-phenotype mapping process to a program. GE is set up such that the evolutionary algorithm is independent of the output programs by virtue of the genotype-phenotype mapping, allowing GE to take advantage of advances in evolutionary algorithm (EA) research. The BNF grammar, like the EA, is a plug-in component of the system that determines the syntax and language of the output code, hence, it is possible to evolve programs in an arbitrary language.

EAs have been used with much success for the automatic generation of programs. In particular, genetic programming (GP) has enjoyed considerable popularity and widespread use [Koza, 1992] [Koza, 1994] [Koza et.al., 1999].

GP originally employed Lisp as its target language, however, many experimenters generate a home grown language, specific to their particular problem.

Grammatical evolution (GE) does not perform the evolutionary process on the actual programs, but rather on variable-length binary strings. A mapping process is employed to generate programs in any language by using the binary strings to select production rules in a Backus Naur Form (BNF) grammar definition. The result is the construction of a syntactically correct program from a binary string that can then be evaluated by a fitness function.

We have named this approach Grammatical Evolution to avoid any confusion with traditional GP, that is, GE adopts a genotype-phenotype distinction and explicitly uses grammars to generate the ouput programs. As stated earlier in Chapter 2, we recommend that a better description for both these approaches is Evolutionary Automatic Programming due to the differences between the various types of GP described previously.

As noted in [Banzhaf, 1994] [Keller & Banzhaf, 1996] and Chapter 3, a mapping process and its subsequent separation of search and solution spaces can result in benefits such as the unconstrained search of the genotype while still ensuring validity/legality of the program's output. For a discussion on various methods of generating legal phenotypes from a genotype see [Yu & Bentley, 1998]. Another potential benefit of such a morphogenic process is that genetic diversity may be enhanced based on the neutral theory of evolution [Kimura, 1983], which states that most mutations driving the evolutionary process are neutral with respect to the phenotype; that is, a mutation may have no effect on the phenotypic fitness of an individual. See Chapter 3 for a more detailed discussion. This phenomenon is facilitated in this system by the use of a degenerate genetic code that is observed in biological genetic systems. This degenerate code facilitates the occurrence of neutral mutations, a consequence of which is that various genotypes can represent the same phenotype, thus facilitating the maintenance of genetic diversity within a population.

This chapter serves as an introduction to GE and the system's unique features, namely the degenerate genetic code and wrapping.

## 4.2 Background

GE is not the first instance in which grammars have been used with evolutionary approaches to automatic programming. A number of other attempts using grammars with GP have been made [Keller & Banzhaf, 1996] [Whigham, 1995a] [Wong & Leung, 1995] [Gruau, 1994] [Horner, 1996] [Paterson & Livesey, 1997] [Freeman, 1998], largely to overcome the so-called "closure" problem, the generation and preservation of valid programs. A more detailed discussion on this area can be found in Chapter 2, Section 2.4.2. As well as examining the closure problem, Whigham [Whigham, 1995a] used grammars as a method to introduce bias into the evolutionary process [Whigham, 1995b] [Whigham, 1996a]. The grammar is used to introduce declarative bias by specifying all the legal statements that can be generated. The grammar could therefore be used to incorporate knowledge by specifying the structure that a solution might take, for example, is it multiline with conditional statements and loops or simply a single line of code. Grammars were also allowed to be modified during runs, thus allowing the system to modify the bias initially specified.

As in Wong and Leung [Wong & Leung, 1995], and Horner [Horner, 1996], derivation trees are used as the genotype representation by Whigham. The derivation trees state exactly which production rules are to be used at any time during the mapping process onto the phenotype. In [Horner, 1996], an implementation of a grammar based GP system is described in which a great deal of effort is put into generating complete derivation trees for the initial generation to ensure the individuals are complete programs and to ensure variation in the size and shapes of solutions. As with all tree-based GP systems, the genetic operators must be designed to maintain closure of the generated programs. Wong and Leung adopted logic grammars, a member of the class of context-sensitive grammars, in which both the terminals and non-terminals can include arguments, thus allowing context-dependency to be enforced.

Paterson & Livesley [Paterson & Livesey, 1997] and later Freeman [Freeman, 1998] attempted to overcome the problem of generating the initial generation by introducing a repair mechanism that used default values in the case that a non-terminal was left without a terminal having been specified. Each uses fixed-length integer arrays as the genotype representation, where each integer represents a production rule from the BNF. When rules cannot be applied in these systems, they are ignored, which can result in a proliferation of introns. Keller & Banzhaf [Keller & Banzhaf, 1996] use a repair mechanism of a different sort, whereby illegal terminal symbols in the generated code are replaced with a legal terminal according to a grammar. Each terminal symbol is represented by a unique binary code; an illegal symbol is replaced by the legal symbol whose code is closest by hamming distance. This system also uses fixed-length genomes, in this case with binary coding.

## 4.3 Grammatical Evolution

GE presents a unique way of using grammars in the process of automatic programming. Variable-length binary string genomes are used, with each codon representing an integer value where codons are consecutive groups of 8 bits in order to make the genetic code degenerate. The integer values are used in a mapping function to select an appropriate production rule from the BNF definition; the numbers generated always representing one of the rules that can be used at that time. GE does not suffer from the problem of having to ignore codon integer values because it does not generate illegal values. The issue of ensuring a complete mapping of an individual onto a program comprised exclusively of terminals is partly resolved using a novel technique to evolutionary algorithms called *wrapping*. This technique draws inspiration from the *overlapping genes* phenomenon exhibited by many bacteria, viruses, and mitochondria that enables them to re-use the same genetic material in the expression of different genes [Lewin, 1999].

GE then, is a system that employs a robust new mapping process, the end result of which is the ability to produce code in any language from a simple binary string. At present, the search element of the system is carried out by an evolutionary algorithm, although conceivably any search method with the ability to operate over variable-length binary strings could be employed. In particular, future advances in the field of evolutionary algorithms can be easily incorporated into this system due to the program representation.

### 4.3.1 The Biological Approach

The GE system is inspired largely by the biological process of generating a protein from the genetic material of an organism. Proteins are fundamental in the proper development and operation of living organisms and are responsible for traits such as eye colour and height [Lewin, 1999].

The genetic material (usually DNA) contains the information required to produce specific proteins at different points along the molecule. For simplicity, consider DNA to be a string of building blocks called nucleotides, of which there are four, named A, T, G, and C, for adenine, tyrosine, guanine, and cytosine respectively. Groups of three nucleotides, called codons, are used to specify the building blocks of proteins. These protein building blocks are known as amino acids, and the sequence of these amino acids in a protein is determined by the sequence of codons on the DNA strand. The sequence of amino acids is very important as it plays a large part in determining the final three-dimensional structure of the protein, which in turn has a role to play in determining its functional properties.

In order to generate a protein from the sequence of nucleotides in the DNA, the nucleotide sequence is first transcribed into a slightly different format, that being a sequence of elements on a molecule known as RNA. Codons within the RNA molecule are then translated to determine the sequence of amino acids that are contained within the protein molecule. The application of production rules to the non-terminals of the incomplete code being mapped in GE is analogous to the role amino acids play when being combined together to transform the growing protein molecule into its final functional three-dimensional form.

The result of the expression of the genetic material as proteins in conjunction with environmental factors is the phenotype. In GE, the phenotype is a computer program that is generated from the genetic material (the genotype) by a process termed a genotype-phenotype mapping. This is unlike the standard method of generating a solution (a program in the case of GE) directly from an individual in an evolutionary algorithm by explicitly encoding the solution within the genetic material. Instead, a many-to-one mapping process is employed within which the robustness of the GE system lies.

Figure 4.1 compares the mapping process employed in both GE and biological organisms.

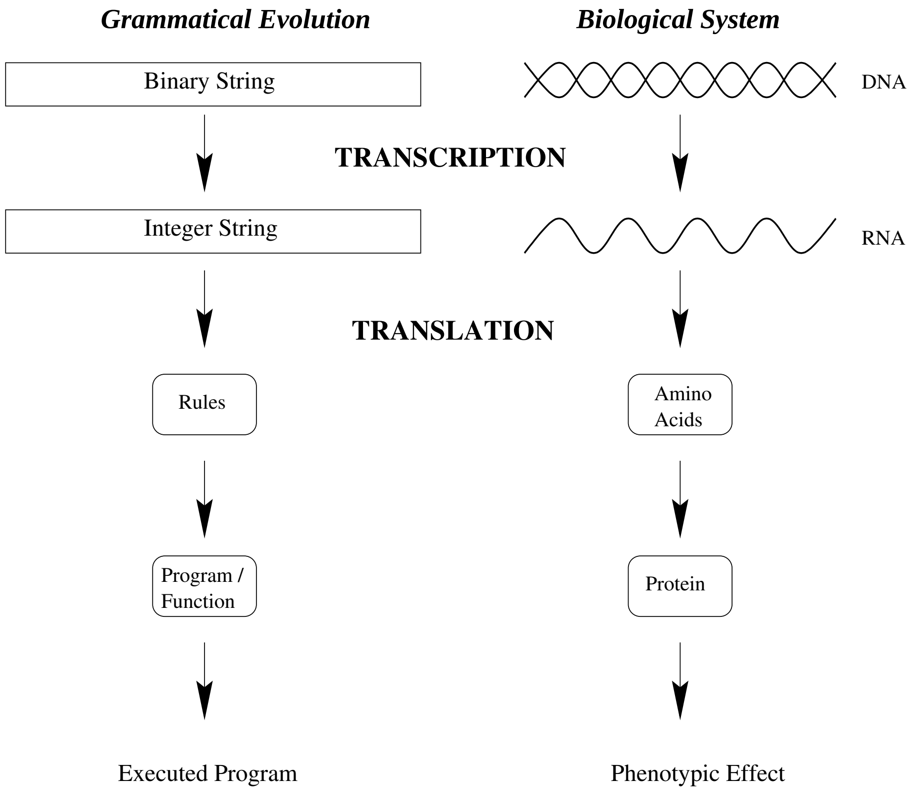


Figure 4.1: A comparison between the grammatical evolution system and a biological genetic system. The binary string of GE is analogous to the double helix of DNA, each guiding the formation of the phenotype. In the case of GE, this occurs via the application of production rules to generate the terminals of the compilable program. In the biological case by directing the formation of the phenotypic protein by determining the order and type of protein subcomponents (amino acids) that are joined together.

### 4.3.2 The Mapping Process

When tackling a problem with GE, a suitable BNF definition must first be decided upon. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand. Complete BNFs are freely available for languages such as C, and these can be plugged into GE.

#### Backus Naur Form

In GE, a BNF definition (see Section 2.4.2) is used to describe the output language to be produced by the system. The compilable code produced will consist of elements of the terminal set $T$. The grammar is used in a developmental approach whereby the evolutionary process evolves the production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely from elements of $T$.

As the BNF definition is a plug-in component of the system, it means that GE can produce code in any language thereby giving the system a unique flexibility.

Below is an example BNF definition, where

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, +, -, /, *, X, 1.0, (, )\}$$

$$S =< expr >$$

And $P$ can be represented as:

```
(A) <expr> ::= <expr> <op> <expr>     (0)
             | ( <expr> <op> <expr> ) (1)
```

```
          | <pre-op> ( <expr> )     (2)
          | <var>                   (3)

(B) <op> ::= +        (0)
           | -        (1)
           | /        (2)
           | *        (3)
(C) <pre-op> ::= Sin
(D) <var> ::= X       (0)
            | 1.0     (1)
```

As noted earlier in Section 2.4.2, there is the possibility of confusion with the terminology adopted with grammars and the GP terms **GPFunctions** and **GPTerminals**. In GE, we use the term **terminals** with its usual meaning in the context of grammars.

For the above BNF, Table 4.1 summarizes the production rules and the number of choices associated with each.

| **Rule no.** | **Choices** |
|---|---|
| A | 4 |
| B | 4 |
| C | 1 |
| D | 2 |

Table 4.1: The number of choices available from each production rule.

### Mapping Process Outline

The genotype is used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = (Codon\ integer\ value)$$

$$MOD$$

$$(Number\ of\ rules\ for\ the\ current\ non-terminal)$$

Consider the following rule i.e., given the non-terminal *op* there are four production rules to select from.

```
(B) <op> :: =     +                  (0)
                | -                  (1)
                | /                  (2)
                | *                  (3)
```

If we assume the codon being read produces the integer 6, then

$$6\ MOD\ 4\ =\ 2$$

would select rule (2) /. Each time a production rule has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

During the genotype-to-phenotype mapping process it is possible for individuals to run out of codons, and in this case we wrap the individual and reuse the codons. This is quite an unusual approach in EAs, as it is entirely possible for certain codons to be used two or more times. This technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [Lewin, 1999].

In GE, each time the same codon is expressed it will always generate the same integer value, but, depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule. What is crucial, however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is the case because the same choices are made each time. However, it is possible that an incomplete mapping could occur, even after several wrapping events, and in this case the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules repeatedly. For example, consider an individual with three codons, all of which specify rule 0 from below,

```
(A) <expr> :: =  <expr><op><expr>        (0)
                |(<expr><op><expr>)      (1)
                |<pre-op>(<expr>)        (2)
                |<var>                   (3)
```

even after wrapping the mapping process would be incomplete and would carry on indefinitely unless stopped. This occurs because the nonterminal <expr> is being mapped recursively by production rule 0, i.e., it becomes <expr><op><expr>. Therefore, the leftmost <expr> after each application of a production would itself be mapped to a <expr><op><expr>, resulting in an expression continually growing as follows:
<expr><op><expr><op><expr><op><expr> etc.

Such an individual is dubbed invalid as it will never undergo a complete mapping to a set of terminals. It is clearly essential that stop sequences are found during the evolutionary search in order to complete the mapping process to a functional program. The stop sequence being a set of codons that result in the non-terminals being transformed into elements of the grammars terminal set.

Beginning from the left hand side of the genome then, codon integer values are generated and used to select rules from the BNF grammar, until one of the following situations arise:

1. A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar.

2. The end of the genome is reached, in which case the *wrapping* operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue, unless an upper threshold representing the maximum number of wrapping events has occurred during this individual's mapping process.

3. In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual is assigned the lowest possible fitness value.

To reduce the number of invalid individuals being passed from generation to generation, a steady state replacement mechanism is employed. One consequence of the use of a steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals.

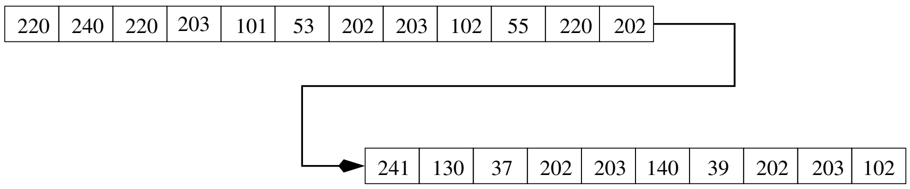


Figure 4.2: An example individual expressed as integers. The integer values are generated by converting the 8-bit binary number that is each codon into its corresponding integer value.

### 4.3.3 Example Individual

Consider the individual in Figure 4.2. These numbers will be used to look up the Table 4.1 that describes the BNF grammar given in Section 4.3.2.

Concentrating on the start symbol <expr>, we can see that there are four productions to choose from.

```
(A) <expr> :: =  <expr><op><expr>        (0)
                |(<expr><op><expr>)      (1)
                |<pre-op>(<expr>)        (2)
                |<var>                   (3)
```

To make this choice, we read the first codon from the chromosome and use it to generate a number. This number will then be used to decide which production rule to use, according to the mapping function given in Section 4.3.2; we have 220 $MOD$ $4 = 0$, meaning we must take the first production (A.0)[1], so that <expr> is now replaced with

```
<expr><op><expr>
```

[1]Production rules are represented here as X.Y, where X represents the non-terminal, and Y represents the rule to be applied to the non-terminal X.

Notice that if this individual is subsequently wrapped each codon will always result in the same integer value, but, depending on the number of choices for the current non-terminal, a different rule number could be selected. In this way, although we have the same codon integer value, it could result in a different physical trait.

Continuing with the first `<expr>`, i.e. always starting from the left-most non-terminal, a similar choice must be made by reading the next codon value (240), and again using the given formula we get 240 $MOD$ 4 = 0, that is rule (A.0). The left-most `<expr>` will now be replaced with `<expr><op><expr>` to give

```
<expr><op><expr><op><expr>
```

Again, we have the same choice for the first `<expr>`, by reading the next codon value 220, the result being the application of rule (A.0) to give

```
<expr><op><expr><op><expr><op><expr><op><expr>
```

Now the left-most `<expr>` will be determined by the codon value 203 which gives us rule (A.3) which is `<expr>` becomes `<var>`. The next codon then determines what value `<var>`, which has two possible production rules, shall take, i.e.

```
(D) <var> ::= X        (0)
            | 1.0      (1)
```

This gives us 101 $MOD$ $2 = 1$ i.e. rule (D.1), which turns out to be **1.0**. We now have the following

```
1.0 <op><expr><op><expr><op><expr><op><expr>
```

The next codon will determine what `<op>` will become, so we have $53\ MOD\ 4 = 1$, which gives a `-`. The next `<expr>` has then to be expanded using the codon value 202, that is $202\ MOD\ 4 = 2$. So we now have

```
1.0-<pre-op><expr><op><expr><op><expr><op><expr>
```

There can only be one outcome for a `<pre-op>` that being **Sin**, therefore, no decision has to be made and so no codon is read. The next `<expr>` is then expanded by the value $203\ MOD\ 4 = 3$ which is rule (A.3), or `<var>` . Its value is then determined by $102\ MOD\ 2 = 0$ , rule (D.0), and the resulting expression is

```
1.0 - Sin(x)<op><expr><op><expr><op><expr>
```

The mapping continues until eventually, we are left with the following expression:

```
1.0 - Sin(x)*Sin(x) - Sin(x)*Sin(x)
```

Notice how all of the codons were required in this case, had there been any extra codons they would have been simply ignored. If we had run out of codons we would have wrapped back to the first codon and continued reading codons until one of the termination conditions, outlined in Section 4.3.2, were met.

### 4.3.4 Genetic Code Degeneracy

Given an 8-bit binary number, each codon in GE can represent 256 distinct integer values, although many of these integer values can represent the same production rule. Taking production rule 2 in Section 4.3.2 as an example, if the current codon value were 8, then $8\ MOD\ 4 = 0$ would select rule (0) $+$.

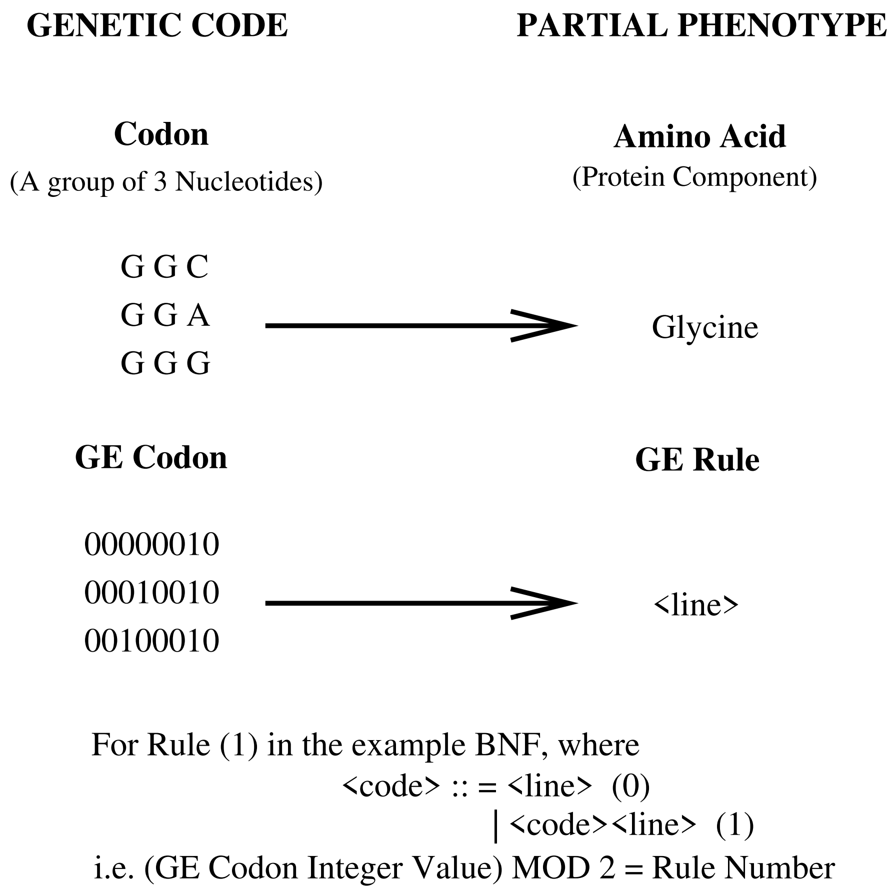


Figure 4.3: Genetic code degeneracy in GE (bottom) compared to the biological genetic code (top). In the case of the biological genetic code, we can see 3 sets of codons representing the same protein component, Glycine. These codons differ only in the base at the third position. In the example GE codons, again we see different codons representing the same production rule from the grammar definition.

The same rule would be chosen if the codon value were 4, 12, 16, etc.

A similar phenomenon can be observed in the genetic code of biological organisms, referred to as *degenerate genetic code* [Lewin, 1999]. Some discussion on this topic is conducted in Chapter 3. There are $4^3$, i.e. 64, unique combinations of nucleotides in a codon, 61 of these coding for a specific amino acid, the other three are special codons that delimit the start and end of genes on the DNA. On average, there are three codons for every amino acid, that is, more than one codon can represent the same amino acid. It has been observed that the first two nucleotides in the codon are often sufficient to specify a particular amino acid, the value of the nucleotide at the third position often being irrelevant. Code degeneracy has interesting implications when it comes to mutation effects. A mutation at the third codon position can often produce what is called a *neutral mutation*, meaning that the amino acid specified will be the same as the one before the mutation event due to the flexibility at the third codon position. With respect to GE, this means that subtle changes in the search space (genotype) may have no effect on the solution space (phenotype), which could result in the maintenance of genotypic diversity throughout a run of the system as different genotypes can represent the same phenotype. It may also preserve valid individuals because the neutral mutations provide a buffering effect against destructive mutation events. Evidence to this effect has been presented in [O'Neill & Ryan, 1999]. More recently advantages of neutrality in the evolution of digital circuits and another GP variant has been discussed [Vassilev & Miller, 2000] [Miller & Thomson, 2000]. Figure 4.3 shows that in the genetic code of biological organisms, the nucleotide at the third position of the codon is independent of the amino acid produced (Glycine). Similarly with GE, it can be seen in the given example that a single bit mutation has no effect on the rule used in this case, e.g.,

$2\ MOD\ 2 = 18\ MOD\ 2 = 34\ MOD\ 2 = 66\ MOD\ 2 = 0$, all specify rule 0. Note, however, if the number of choices in the example was uneven, e.g. three, a single bit mutation would effect the rule used.

Kimura's neutral theory of evolution [Kimura, 1983] suggests that it is these neutral mutations which are responsible for the genetic diversity that has been observed in natural populations, a feature that has been exhibited within GE.

### 4.3.5 The Search Algorithm

We adopt an evolutionary algorithm to perform a search of the program solution space by indirectly operating on variable-length, binary strings. Conceivably any search algorithm that can operate on binary strings could employ GE's mapping process to generate a program. As the population being evolved comprises simple binary strings, we do not have to employ any special crossover or mutation operators, and an unconstrained search is performed on these strings due to the genotype-to-phenotype mapping process that will generate syntactically correct individuals.

The evolutionary algorithm adopted in this case is a variable-length genetic algorithm. Individual initialisation is achieved by randomly generating variable-length binary strings within a pre-specified range of codons. For all experiments conducted in this paper we use the initialisation range of one to ten codons, where a codon is a group of eight bits.

As well as the standard genetic operators of mutation (point) and crossover (one-point) we adopt a codon duplication operator. Duplication involves randomly selecting a number of codons to duplicate and the starting position of the first codon in this set. The duplicated codons are placed at the penultimate codon position at the end of the chromosome so as to facilitate their incorporation into the phenotype. We do this because if the the individual produced a completely mapped program after reading the last codon, and we placed duplicated codons after this point, they will not be used by this individual, until some other genetic operator allowed them to be switched on.

The GE component of the system, that is, the part that carries out the mapping from binary string to the output code, could conceivably be plugged-in to the fitness function of any EA. The result of this is that GE can benefit from the latest advances in EA research, for example, future investigations will be conducted into the use of competent GAs that have been shown to have superior scaling properties to the simple GA [Goldberg et.al, 1989] [Thierens, 1999] [Harik & Goldberg] [Harik, 1999] [Kargupta, 1998] [Pelikan et. al., 2000].

## 4.4 Discussion

In Chapter 3 we described a number of desirable features that an evolutionary automatic programming system could avail of based on our lessons from Molecular Biology. Given the description above of GE, it can be seen that we have taken advantage of a number of these features, i.e. the first six of eight. Specifically,

1. We achieved a generalised encoding that can represent a variety of structures. We do this by using a separation of the search and solution spaces through the use of a genotype-phenotype distinction. The transformation or mapping process allows us to use a BNF grammar definition as a plug-in component of the system, the BNF being a generalised encoding of the structures we are evolving, i.e. programs. This

allows us to generate programs in an arbitrary language. The genotype-phenotype distinction in turn facilitates the following five features.

2. The potential of efficiency gains for the evolutionary search have been achieved with the use of a degenerate genetic code. The existence of the degenerate genetic code means that neutral mutation events are possible, and therefore neutral evolution can occur. As discussed in Chapter 3, this is reported to have benefits for the efficiency of the evolutionary search.

3. Following on from the existence of the degenerate genetic code, the potential for the maintenance of genetic diversity exists. In the above system description 8-bit codons are used to represent integers values. A property of the mapping function is that it is redundant, giving rise to what we call the degeneracy property, thus different integer values can represent the same grammar production rule. Consequently, different genotypes can represent the same phenotype, thus facilitating genetic diversity within a population.

4. Preservation of functionality is also aided by virtue of the degenerate genetic code. As discussed in Chapter 3, neutral mutation events can occur that change the genotype but may not result in a phenotypic change.

5. With the introduction of the wrapping operator we have provided a facility whereby re-use of genetic material is possible. In the event that an individual is incompletely mapped upon reaching the last codon on a genome, the wrapping operator allows the re-use of codons at the beginning of the genome.

6. A compression of representation can be facilitated as a direct result of the wrapping operator. An investigation into this fact will be conducted in Chapter 6.

Investigations as part of an analysis of GE, conducted in Chapter 6, will demonstrate the existence of features 2, 3, 4, 5 and 6.

## 4.5 Conclusions

We have presented the Grammatical Evolution (GE) system and described its operation and unique features. In the next chapter we present proof of concept problems that demonstrate GE's ability to generate code in an arbitrary language and also conduct a comparison with GP.

Chapters 6 and 7 will then report on an analysis of GE, including the degenerate genetic code, the wrapping operator, and the role that crossover plays in the system.

# Chapter 5

# Proof of Concept

## 5.1 Introduction

We now describe the application of GE to a number of problem domains, namely a symbolic regression problem, a symbolic integration problem, the Santa Fe ant trail, and the evolution of caching algorithms. We use these problems to illustrate that GE is indeed capable of producing compilable code. These problems are deliberately diverse; for example, in the symbolic regression problem a simple one-line expression is evolved, whereas in the Santa Fe trail problem a multiline function including branch statements is required. A description of each problem domain used now follows, including a report on the performance of GE, and a comparison with results obtained by GP for these problem domains is also provided.

## 5.2 Symbolic Regression

Symbolic regression problems involve finding some mathematical expression in symbolic form that represents a given set of input and output pairs.

The aim is to determine the function that maps the input pairs onto the output pairs. The particular function examined is

$$f(X) = X^4 + X^3 + X^2 + X$$

with the input values in the range $[-1..1]$.

The grammar used in this problem is given below:

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, Exp, Log, +, -, /, *, X, 1.0, (, )\}$$

$$S =< expr >$$

And $P$ can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>       (0)
             | ( <expr> <op> <expr> )   (1)
             | <pre-op> ( <expr> )      (2)
             | <var>                    (3)


(2) <op> ::= +        (0)
           | -        (1)
           | /        (2)
           | *        (3)


(3) <pre-op> ::= Sin     (0)
               | Cos     (1)
               | Exp     (2)
               | Log     (3)
```

| | |
|---|---|
| Objective : | Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 $(x_i, y_i)$ data points, where the target functions is the quartic polynomial $X^4 + X^3 + X^2 + X$ |
| GPTerminal Set: | $X$ (the independent variable) |
| GPFunction Set | $+, -, *, \%, \sin, \cos, \exp, \log$ |
| Fitness cases | The given sample of 20 data points in the interval $[-1, +1]$ |
| Raw Fitness | The sum, taken over the 20 fitness cases, of the error |
| Standardised Fitness | Same as raw fitness |
| Hits | The number of fitness cases for which the error is less than 0.01 |
| Wrapper | None |
| Parameters | $M = 500$, $G = 51$ |

Table 5.1: A Koza-style tableau for Symbolic Regression

```
(4) <var> ::= X       (0)
            | 1.0     (1)
```

The production rules for < *expr* > are similar to those given earlier in Section 4.3.2, with the terminal operator set also including Cos, Exp, and Log.

The fitness for this problem is given by the sum, taken over 20 fitness cases, of the error between the evolved and target functions. Tableaus for GP and GE can be seen in Tables 5.1 and 5.2, respectively.

| | |
|---|---|
| Objective : | Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 $(x_i, y_i)$ data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$ |
| Terminal Operands: | $X$ (the independent variable), 1.0 |
| Terminal Operators | The binary operators $+, *, /$, and $-$<br>The unary operators Sin, Cos, Exp and Log |
| Fitness cases | The given sample of 20 data points in the interval $[-1, +1]$<br>i.e. { -1, -.9, -.8, -.76, -.72, -.68, -.64, -.4, -.2, 0, .2, .4, .63, .72, .81, .90, .93, .96, .99, 1 } |
| Raw Fitness | The sum, taken over the 20 fitness cases, of the error |
| Standardised Fitness | Same as raw fitness |
| Wrapper | Standard productions to generate C functions |
| Parameters | Population Size = 500,<br>Termination when Generations = 51<br>Prob. Mutation = 0.01, Prob. Crossover = 0.9<br>Prob. Duplication = 0.01, Steady State |

Table 5.2: Symbolic Regression Tableau for GE

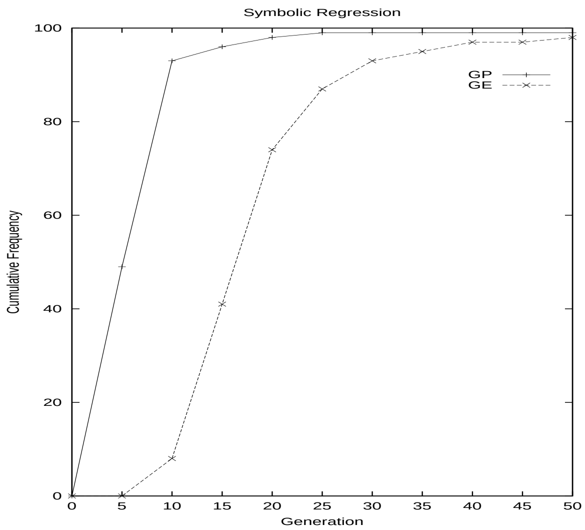


Figure 5.1: Cumulative frequency of success measure of GE versus GP on the symbolic regression problem.

### 5.2.1 Results

GE successfully found the $X+X^2+X^3+X^4$ target function. A cumulative frequency measure of success over 100 runs can be seen in Figure 5.1, along with a similar measure for GP, and in this case GP outperforms GE.

## 5.3 Symbolic Integration

Symbolic integration involves finding a function that is the integral of the given curve. Similarly to symbolic regression the system is given a set of input and output pairs and must determine the function that maps one onto the other.

The particular function examined was

$$f(X) = Cos(X) + 2X + 1$$

with the input values in the range $[0..2\pi]$ and the target integral curve being

$$f(X) = Sin(X) + X + X^2$$

In a manner similar to Koza [Koza, 1992], we reduce the problem to symbolic regression by integrating the function examined and performing symbolic regression on the target integral curve. The fitness for this problem is given by the sum, taken over 20 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point $x_i$ and the value of the numerical integral $\mathbf{I}(x_i)$.

The grammar used for this problem is the same as for the symbolic regression problem given in section 5.2, and a tableau for GE is given in Table 5.3. For comparison the corresponding GP tableau is included in Table 5.4.

### 5.3.1 Results

GE successfully found the target integral function $Sin(X)+X+X^2$ illustrating that useful expressions could be generated by the system. A cumulative frequency measure of success over 100 runs can be seen in Figure 5.2. The same problem was tackled using GP and a cumulative frequency of success for 100 runs can be seen in Figure 5.2. As can be seen GE outperforms GP on this problem from around the $10^{th}$ generation.

When comparing GE to GP, it is important to note how the initial generation is formed in each system. In the case of the GP system a ramped half-and-half generation mechanism is used, creating a range of individuals

| | |
|---|---|
| Objective : | Find a function, in symbolic form, that is the integral of a curve presented either as a mathematical expression or as a given finite sample of points $(x_i, y_i)$ |
| Terminal Operands: | $X$ (the independent variable) |
| Terminal Operators | The binary operators $+, *, /, -$ and the unary operators Sin, Cos, Exp and Log |
| Fitness cases | A sample of 20 data points in the interval $[0, 2\pi]$ |
| Raw Fitness | The sum, taken over the 20 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point $x_i$ and the value of the numerical integral $\mathbf{I}(x_i)$ |
| Standardised Fitness | Same as raw fitness |
| Wrapper | Standard productions to generate C functions |
| Parameters | Population Size = 500,<br>Termination when Generations = 51<br>Prob. Mutation = 0.01, Prob. Crossover = 0.9<br>Prob. Duplication = 0.01, Steady State |

Table 5.3: Symbolic Integration Tableau for GE

| | |
|---|---|
| Objective : | Find a function, in symbolic form, that is the integral of a curve presented either as a mathematical expression or as a given finite sample of points $(x_i, y_i)$ |
| GPTerminal Set: | $X$ (the independent variable) |
| GPFunction Set | $+, -, *, \%, \sin, \cos, \exp, \log$ |
| Fitness cases | The given sample of 50 data points in the interval $[0, 2\pi]$ |
| Raw Fitness | The sum, taken over the 50 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point $x_i$ and the value of the numerical integral $\mathbf{I}(x_i)$ |
| Standardised Fitness | Same as raw fitness |
| Hits | The number of fitness cases for which the error is less than 0.01 |
| Wrapper | None |
| Parameters | $M = 500$, $G = 51$ |

Table 5.4: A Koza-style tableau for Symbolic Integration

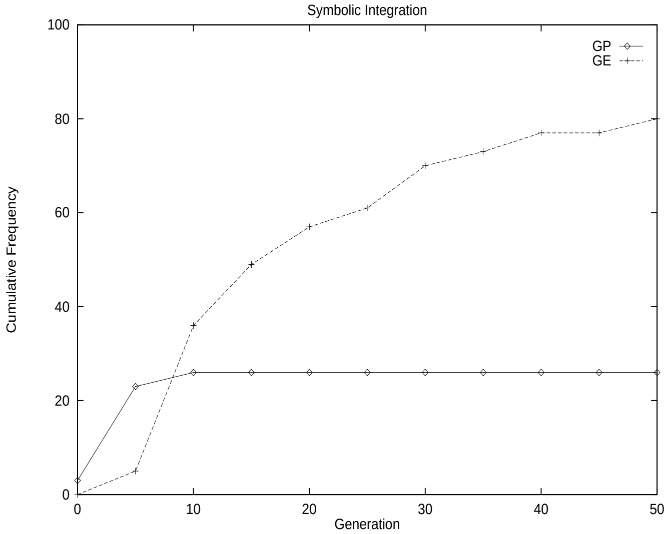


Figure 5.2: Cumulative frequency of success measure of GE versus GP on the symbolic integration problem.

of varying shapes and depths, whereas with GE the generation is totally random. Also, every individual in the initial population is engineered to be unique in the GP system, whereas with GE this is not the case. In the case of this problem we feel that the generation strategy for the initial generation is providing GP with an advantage due to the regular nature of the solution, as illustrated in Figure 5.3, and by the number of successful solutions in early generations, see Figure 5.1. Additional support is provided by the fact that on the symbolic integration problem the solution does not have the same regularity as is the case here, see Figure 5.4.

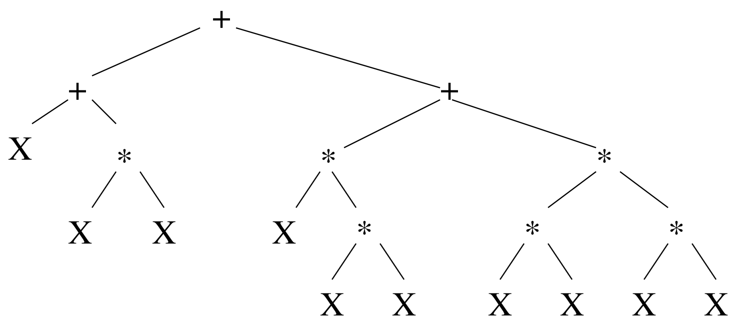


Figure 5.3: Example solution to Symbolic Regression as a GP parse tree. Note the regular, repetitive nature of the solution.

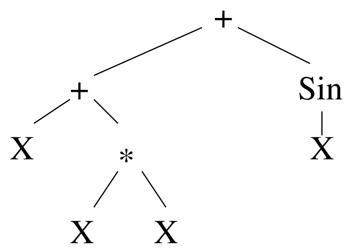


Figure 5.4: Example solution to Symbolic Integration as a GP parse tree.

## 5.4 Santa Fe Ant Trail

The Santa Fe ant trail is a standard problem in the area of GP and can be considered a deceptive planning problem with many local and global optima [Langdon & Poli, 1998]. The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a non-continuous trail within a specified number of time steps, the trail being located on a 32 by 32 toroidal grid. The ant can only turn left, right, move one square forward, and may also look ahead one square in the direction it is facing to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid facing the first piece of food on the trail. The grammar used in this problem is different to the ones used for symbolic integration and symbolic regression. In this problem we wish to produce a multiline function, as opposed to a single line expression. The grammar used is given below:

$$N = \{code, line, expr, if-statement, op\}$$

$$T = \{left(), right(), move(), food\_ahead(),$$

$$else, if, \{, \}, (, ), ;\}$$

$$S =< code >$$

And $P$ can be represented as:

```
(1) <code> :: =  <line>              (0)
                |<code><line>        (1)


(2) <line> :: =  <if-statement>      (0)
                |<op>                (1)


(3) <if-statement> :: = if(food_ahead())
                          {<line>}
                        else
                          {<line>}


(4) <op> :: =    left();              (0)
               | right();             (1)
               | move();              (2)
```

Note that it is the rules for the non-terminal `<code>` that are responsible for the production of multiline code. A tableau describing this problem can be seen in Table 5.5.

| | |
|---|---|
| Objective : | Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail. |
| Terminal Operators: | left(), right(), move(), food_ahead() |
| Terminal Operands: | None |
| Fitness cases | One fitness case |
| Raw Fitness | Number of pieces of food before the ant times out with 615 operations. |
| Standardised Fitness | Total number of pieces of food less the raw fitness. |
| Wrapper | None |
| Parameters | Population Size = 500,<br>Termination when Generations = 51<br>Prob. Mutation = 0.01, Prob. Crossover = 0.9<br>Prob. Duplication = 0.01, Steady State |

Table 5.5: Grammatical Evolution Tableau for the Santa Fe Trail

### 5.4.1 Results

GE was successful at finding a solution to this case of the Santa Fe trail, demonstrating that GE can generate multiline code by using a simple modification to the grammar definition. An example solution produced by GE is:

```
move();
left();
if(food_ahead())
    left();
else
    right();
right();
if(food_ahead())
    move();
else
    left();
```

which is executed in a loop until the number of time steps allowed is reached. A cumulative frequency measure of success over 100 runs of GE can be seen in Figure 5.5. This figure also shows the performance of the GP system, which incorporated solution length in the fitness measure, as well as the number of pieces of food picked up, as follows:

$$Fitness = Food\ Picked\ Up\ + \frac{(Solution\ Length)^2}{1000}$$

The fitness of a GE individual, on the other hand, was simply the number of food pieces picked up. The performance of the two systems is comparable in this case with GP slightly outperforming GE over the first 30 generations.

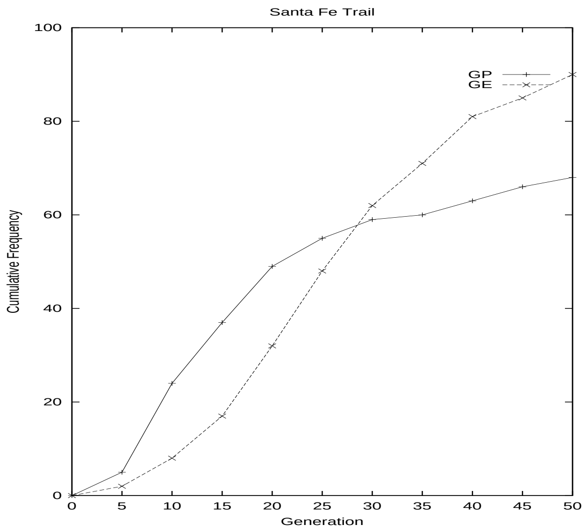


Figure 5.5: The cumulative frequency of success measure for GE versus GP on the Santa Fe trail problem. The results show the case when GP uses solution length in the fitness function (GE has no such measure).

In order to see how the GP system would perform without the solution length as a measure of fitness, we ran 100 more runs of the GP system removing the solution length measure. We felt that using solution length as a measure of fitness required a prior knowledge of the solution's length and therefore gave an unfair advantage, as GE did not use such a measure. Figure 5.6 shows a comparison of the cumulative frequency measure of success for these results with the results produced by GE. As can be seen from the figure the performance of the GP system was compromised and as a result GE outperformed GP.

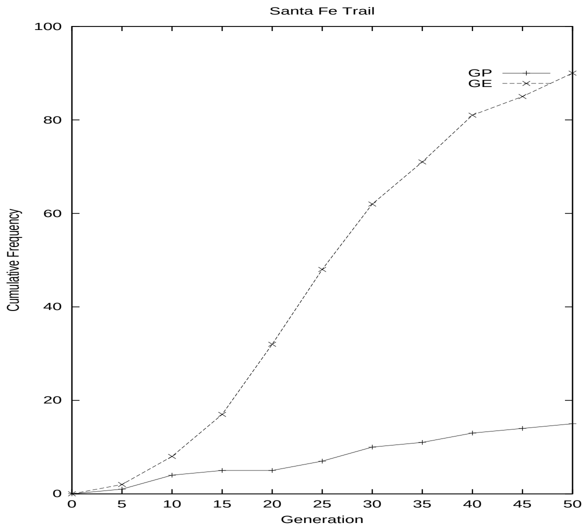


Figure 5.6: The cumulative frequency of success measure for GE versus GP on the Santa Fe trail problem. The results shown illustrate the case where the solution length constraint is removed from GP.

## 5.5 Caching Algorithms

We now describe how GE was applied to the real world problem of evolving a Caching Algorithm, at which GP has been found to generate algorithms that did not perform as well as those designed by humans
[Paterson & Livesey, 1997].

A cache memory is a small, fast memory, ideally into which the *address space* of a running program is held in order to improve efficiency of the program's execution. Caches operate on the notion of *locality of reference*; that is, if a word in memory has recently been accessed, it is likely to be required again soon. The executing program generates a stream of *requests* of main memory locations to access. A recording of this request stream is referred to as a *trace*. The cache has a fixed number of *lines* each of which can hold a word (an address). When a *request* is met from the cache memory this is called a *hit*, otherwise a *miss* occurs. In the event of a miss, the requested memory location is copied into the cache to the first empty *line* available, otherwise a caching algorithm determines which line of the cache will be replaced (the *victim*). In order that the caching algorithm can make an educated decision as to which line will become the victim, typically some management information is available, e.g. an ordered list of lines.

For the purposes of these experiments, we adhered as closely as possible to the procedures outlined in [Paterson & Livesey, 1997], in order to facilitate a comparison of the systems. Thus, the same trace files, a similar grammar (no default or multiple rules were used) with the same terminal set, an integer management array $info[\,]$ (one element per line), and the same fitness function were used.

The raw fitness of a caching algorithm is therefore given as

$$number\ of\ runs - number\ of\ misses$$

where a run is a consecutive series of identical requests.

The terminal operators and their respective functions are given in Table 5.6.

| **Terminal Operator** | **Function** |
|---|---|
| $write\_x(i, v)$ : | sets info[i] to v |
| $read\_x(i)$ : | returns info[i] |
| $small\_x(i, v)$ : | index of smallest element of info[] |
| $large\_x(i, v)$ : | index of largest element of info[] |
| $random\_x(i, v)$ : | index of random element of info[] |
| $counter()$ : | successive values 0, 1, 2 etc |
| $div(x, y)$ : | if y==0 then 1 else x / y |
| $rem(x, y)$ : | if y==0 then 1 else x % y |

Table 5.6: Available terminal operators.

The grammar as adopted in most of the experiments described in the next section is as follows:

```
(1)  <stmts> :: = <stmt>
                | <stmt>;<stmts>

(2)  <stmt> :: = if(<expr>){<stmts>;}else{<stmts>;}
               | write_x(<expr>,<expr>);
               | victim=<expr>;
```

```
(3)  <expr> :: = <term>
               | <term>+<term>
               | <term>-<term>
               | <term>*<term>
               | div(<term>,<term>)
               | rem(<term>,<term>)

(4)  <term> :: = CACHESIZE
               | <num>
               | <fun>
               | (<expr>)

(5)  <num> :: = <mant>
              | <mant><zeros>

(6)  <mant> :: = 0 | 1 | 2 | 5

(7)  <zeros> :: = 0
                | 0<zeros>

(8)  <fun>  :: = counter()
               | read_x(<expr>)
               | small_x()
               | large_x()
               | random_x()
```

### 5.5.1 Results

Similar to Paterson & Livesley [Paterson & Livesey, 1997], we employ three trace files from Flanagan [Flanagan, 1992]. One file is used for training GE, while the resulting algorithms are tested using the other two. The training file `ken2.00200` contains around 400,000 requests with about 250,000 distinct addresses. Paterson & Livesley reduced memory requirements by eliminating the lower order 10 bits of each addresses, thus reducing the alphabet size to just 500, i.e. 500 distinct addresses as opposed to 250,000 originally. We use the files in their original state, and show comparisons with Paterson & Livesley's algorithm below. Paterson & Livesley found that the speed of the cache simulator frustrated his attempts to run many experiments, and we suffered the same experience. To alleviate this, we use a population of just 50 individuals, compared to his 500. For each of the experiments that follow, ten runs were conducted, and the best individual at the end of twenty generations was analysed.

**Small Cachesize**

The first experiment consists of using a cache size of just 20, clearly, it is very difficult to exploit any locality of reference with such a small size. The best individual generated by GE was

```
GE 1 :        victim = counter() - CACHESIZE;
```

This curious looking individual simply cycles through the cache when searching for a victim, selecting it without consideration for how recently it was used. There is, however, an implicit temporal quality about the manner in which victims are selected. A word is guaranteed to remain in the cache for at least $CACHESIZE$ requests, longer if it (or other cache members) is requested. This outperforms the human designed Least Recently Used algorithm (LRU) by 15.7%, and, by virtue of its simplicity, is far less expensive to use.

**Large Cachesize**

A cachesize of 200 lines was used in this instance, and one of the best individuals produced was again GE 1. However, in this case the number of misses was far less, outperforming LRU by 46.2%.

**Using info** [ ]

Given the level of disinterest exhibited by GE to the use of the info[] array, we ran an experiment that forced it to write into some element before choosing its victim. The grammar in this case varied from the one outlined above in the case of the production rules for the non-terminals `stmts` and `stmt`. Both of these rules were replaced with the following:

```
(1)  <stmts> :: = write_x(<expr>,<expr>); victim=<expr>;
```

The best of run individual generated by this system was:

```
GE 2 :   write_x( CACHESIZE + counter(), CACHESIZE + counter())
         victim = CACHESIZE + counter();
```

Rather predictably, this individual simply ignores what it writes into the array, and promptly chooses its victim using the same method as in the other experiments. In fact, the ubiquity of the beautifully simple $CACHESIZE + counter()$ algorithm is demonstrated by virtue of its appearance in no less than three parts of this algorithm.

### Constants

Paterson & Livesley reported on an experiment where the use of constants was prohibited. He was pleased to note that imposing this restriction actually improved the performance of their system. In no case did GE produce an individual that required constants, and it appears that the system discovered that constants were unnecessary in this case.

### Comparison of Algorithms

A comparison between the LRU algorithm and those which had been evolved was carried out by rating their performance on the other two trace files, and the number of misses obtained in each case can be seen in Table 5.7.

| **Algorithm (Cachesize)** | **ken2.00100 (Misses)** | **ken2.00200 (Misses)** | **Average of % Improvement over LRU** |
|---|---|---|---|
| LRU (20) | 374,596 | 380,041 | - |
| LRU (200) | 367,104 | 373,935 | - |
| GE1 (20) | 300,569 | 318,444 | 17.97 |
| GE1 (200) | 106,067 | 82,856 | 74.51 |
| GE2 (20) | 300,569 | 318,445 | 17.97 |
| GE2 (200) | 106,068 | 82,855 | 74.51 |

Table 5.7: Algorithm performance comparison.

Paterson & Livesley were dismayed to report that the algorithms they produced were very poor at generalising to unseen data, and that they suffered greatly from overfitting. The algorithms produced by GE don't suffer from this problem, and perform extremely well on the new data.

One factor that might contribute to the large difference in performance was Paterson & Livesley's decision to simplify the problem - the problem as posed to GE was possibly too large to spot any patterns in.

The solution generated by GE was so general that when applied to a larger cache size, a huge increase in performance was observed. When the cache size was increased by an order of magnitude for LRU, the accompanying increase in performance shows that the associated cost of such an increase would probably not pay off.

**Simplified Stream Data**

Another set of experiments were conducted to determine if using the simplified data stream as adopted in [Paterson & Livesey, 1997] would have a detrimental effect on the generalisation capabilities of solutions found by GE. Similarly to Paterson & Livesley we removed the low-order 10 bits from each address in the training stream file, thus reducing the alphabet size to 500 as opposed to 250,000 in the original stream.

Experiments with the small cachesize, large cachesize and forcing the solutions to write to the info[] array were conducted, and in each case GE re-discovered the solutions found in the original experiments. Simplifying the training data set thus had no effect on the generalisation capabilities of GE.

## 5.6 Conclusions

We have presented a number of proof of concept problems that have been successfully tackled by GE. The reported results demonstrate GE's ability to generate compilable code in any language using its unique genotype-phenotype mapping process. In accordance with the No Free Lunch theorem, we do not expect GE to be good at all problems. However, we have demonstrated GE's ability to solve a diverse set of problems, and a comparison of performance to GP, has shown GE to be on a par, and in some cases superior to GP. We now proceed by conducting an analysis of GE on some of the problem domains reported in this chapter.

# Chapter 6

# Analysis of Grammatical Evolution

## 6.1 Introduction

In this chapter we analyse some of the unique features of GE using two problem domains described earlier in Chapter 5, namely the Santa Fe ant trail and symbolic regression. In particular, our focus is turned to the wrapping operator and the degenerate genetic code.

Wrapping exists in GE mainly for the purpose of completion of an otherwise incomplete genotype-phenotype mapping. That is, in the event that the code being generated is still comprised of non-terminal symbols from the grammar, and the end of the genome has been reached, wrapping attempts to complete the mapping by allowing the re-use of the genome.

The degenerate genetic code is present in GE due to the modulo function that is adopted by the mapping function when selecting an appropriate production rule to be applied to the current non-terminal in the code being generated. We wish to establish if the degenerate genetic code could be playing a role in the maintenance of genetic diversity within our populations as has been observed in [Banzhaf, 1994]. Kimura [Kimura, 1983] has claimed that neutral evolution is a candidate that could be responsible for the genetic diversity that is observed in biological populations, and could be enabled by virtue of the existence of the degenerate genetic code in GE. Recent findings suggest that the existence of a genetic code with a variable degree of degeneracy could allow a more efficient evolutionary search, resulting in greater fitness being attainable with higher degrees of degeneracy [Engelhardt & Newman, 1998].

A number of experiments are carried out to determine the utility and effects of the wrapping feature, and to elucidate whether the degenerate code is having an effect on the system's performance and genetic diversity. We firstly describe those experiments pertaining to the wrapping operator.

## 6.2 Wrapping Operator

Two sets of experiments for each problem domain are carried out with each test comprising 100 runs. In the first test the wrapping feature is enabled, and disabled in the second.

### 6.2.1 Results

The results show that wrapping occurs, particularly in the earlier stages of runs, providing evidence that GE exploits this feature. The average number of individuals undergoing wrapping at each generation can be seen in Figure 6.1, and over time it can be seen that the number of individuals undergoing wrapping decreases.

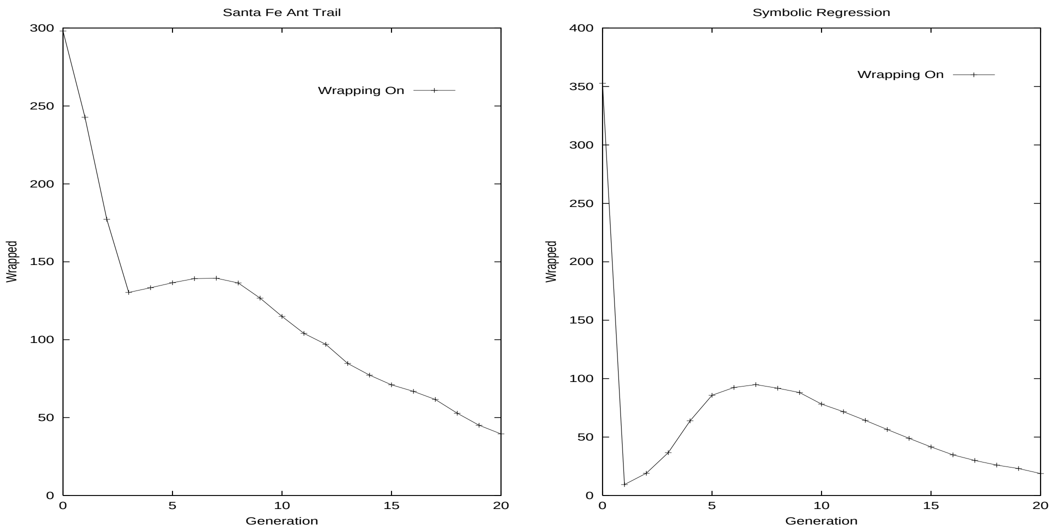


Figure 6.1: Number of individuals wrapped on the symbolic regression and Santa Fe trail problems.

### Invalid Individuals

Looking at the number of invalid individuals (Figure 6.2) we can see that there is a decrease in their numbers both in the presence and absence of wrapping over the initial generations.

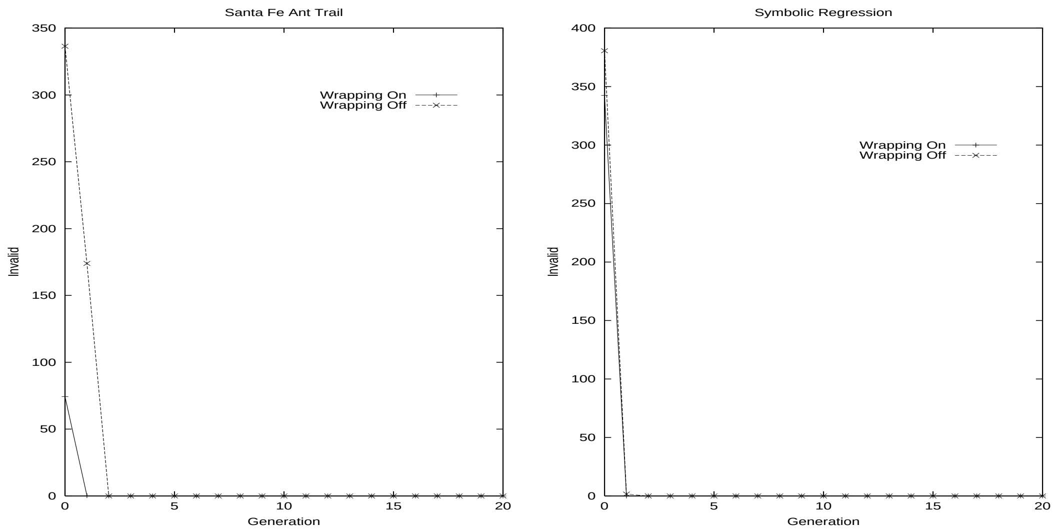


Figure 6.2: The number of invalid individuals for each generation in the presence and absence of wrapping on the symbolic regression and Santa Fe trail problems. This figures illustrates that all invalid individuals are removed from the population early on in runs, even without wrapping being present, due to the replacement mechanism.

### Cumulative Frequency of Success

The effect of wrapping or otherwise on the cumulative frequency of success measures can be seen in Figure 6.3. As can be seen, there appears to be no effect on the success rates on the symbolic regression problem. However, in the case of the Santa Fe trail, there is a decrease in the cumulative frequency of success when wrapping is disabled.

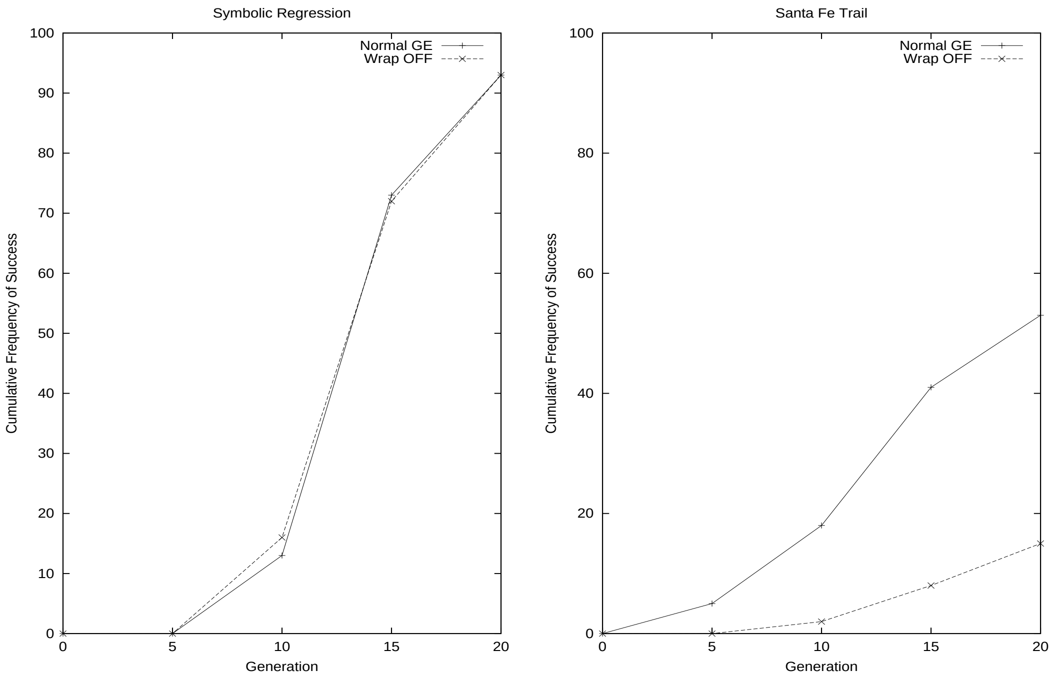


Figure 6.3: Figure shows the cumulative frequency of success measures on both problems with and without the presence of wrapping.

### Genome Lengths

A comparison of the average actual genome lengths to the average effective genome lengths is given in Figure 6.4. The effective genome length is a measure of the number of expressed genes during the genotype-to-phenotype mapping process, while the actual length is the number of genes on the chromosome. The actual length is, on average, longer than the effective length for these problems, and when wrapping is turned off the actual genome lengths increase on the symbolic regression problem. On the Santa Fe trail problem, the actual lengths increase initially when compared to the values in the absence of wrapping, but as time progresses the actual genome lengths are greater in the presence of wrapping.

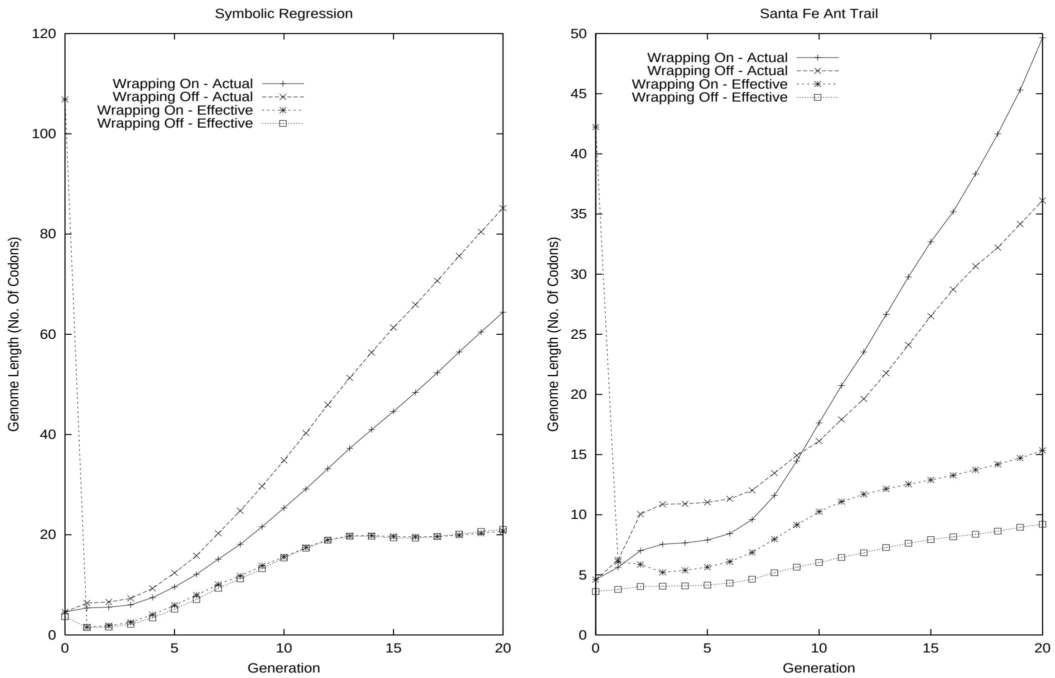


Figure 6.4: The figure shows the actual versus effective genome length for symbolic regression and the Santa Fe trail in the presence and absence of wrapping.

For symbolic regression, this illustrates that the wrapping mechanism reuses the codons that are already present on the genome, in effect compressing the program into a smaller number of codons to when the wrapping feature is enabled, by re-using the genetic material already present on the genome rather than extending its physical length.

### 6.2.2 Discussion

In the symbolic regression problem it is noted that while genome lengths are constrained when the wrapping feature is enabled, there is little effect on the cumulative frequency of success.

The fact that the wrapping feature is important to finding solutions to the Santa Fe trail in 20 generations, and no ill effects are demonstrated in performance on the symbolic regression problem when wrapping is disabled, suggests that wrapping is a useful feature to have available.

The dependency of the Santa Fe trail's solution success rate on wrapping can be attributed to the regular, repetitive nature of the required solution that can be exploited effectively by the wrapping mechanism, particularly earlier on in runs. Solutions are comprised of sequences of the operations $left$, *right*, and *move* with some conditional statements that allow the ant to search for the next piece of food. This search can be achieved through a repetitive application of these operators.

A comparison of the number of invalid individuals produced in both problem domains examined here shows that invalid individuals are restricted almost exclusively to the initial generations (Figure 6.2). The effective removal of these unwanted individuals could be attributed to the selection and replacement mechanisms used in the evolutionary algorithm, in particular the steady state replacement strategy. This is an example of the effective use of the illegal individual replacement constraint (i.e. the removal of incompletely mapped individuals in the case of GE) to evolve legal phenotypes, as described in [Yu & Bentley, 1998].

In summary, wrapping is shown to be useful in improving the success rate of runs, and is also responsible for constraining genome lengths.

## 6.3 Degenerate Genetic Code

In this case six sets of experiments are carried out for each problem domain. The first set comprises 100 runs in which genetic code degeneracy is removed as far as possible by reducing the number of bits in a codon to the lowest possible value that can still represent the maximum number of productions rules belonging to any one non-terminal. The second set is of another 100 runs in which degeneracy is present as in the standard GE implementation using 8-bit codons [O'Neill & Ryan, 1999].

The third set of 100 runs consider the situation where the degeneracy is increased above the standard GE implementation, in this case with the adoption of 16-bit codons.

In addition, a fourth, fifth and sixth set of runs are conducted to take into consideration degrees of degeneracy between 16 and 8-bits (12-bits), and between 8-bits and the case where degeneracy is removed (6-bits, and 4-bits).

In the standard GE implementation each codon can represent 256 distinct integer values. Therefore, in this state GE can represent up to 256 productions for each non-terminal in the grammar. In the case of the Santa Fe trail the maximum number of productions that any one non-terminal has is 3, and for symbolic regression problems this number is 4. As a result, the minimum number of bits any codon can have in the case of these problems is two, as this can represent a maximum of 4 distinct productions. It follows that in the case of the symbolic regression problem, all degeneracy has been removed, while it still exists to a small extent in the Santa Fe trail problem.

We wish to examine the effects that the degenerate code has on the genetic diversity in our population, and we do this by examined two measures of diversity. The first is termed the *mean variety* and is obtained by calculating the average of the variances at each bit locus on the genome, and is given by the following formula:

$$Mean\ Variety = \frac{\sum\limits_{1}^{L} \frac{\sum\limits_{i=1}^{N}(X_i-\mu)^2}{N}}{L}$$

where $X_i$ is the locus value at position $i$, $\mu$ is the mean of $X_i$ at locus $i$, $N$ is the sample population size, and $L$ is the number of loci analysed.

With a population size of 500, the maximum mean variety value obtainable is 0.25. The value 0.25 corresponds to those populations with a greater degree of variance, whereas those populations with values tending towards zero have lower degrees of variance. This measure attempts to establish how different the individual genotypes in any given population are. The second measure of variety is simply the number of unique individuals in a population, and can be used to some extent to illustrate the genetic diversity within a population [Langdon, 1998].

### 6.3.1 Results

A cumulative frequency measure in the presence and absence of code degeneracy can be seen in Figure 6.5 that shows a superior performance on the Santa Fe trail in the presence of a degenerate code. The difference in the case of the easier symbolic regression problem is not as clear, indeed both 12 and 16-bit do not perform as well as when degeneracy is switched off.

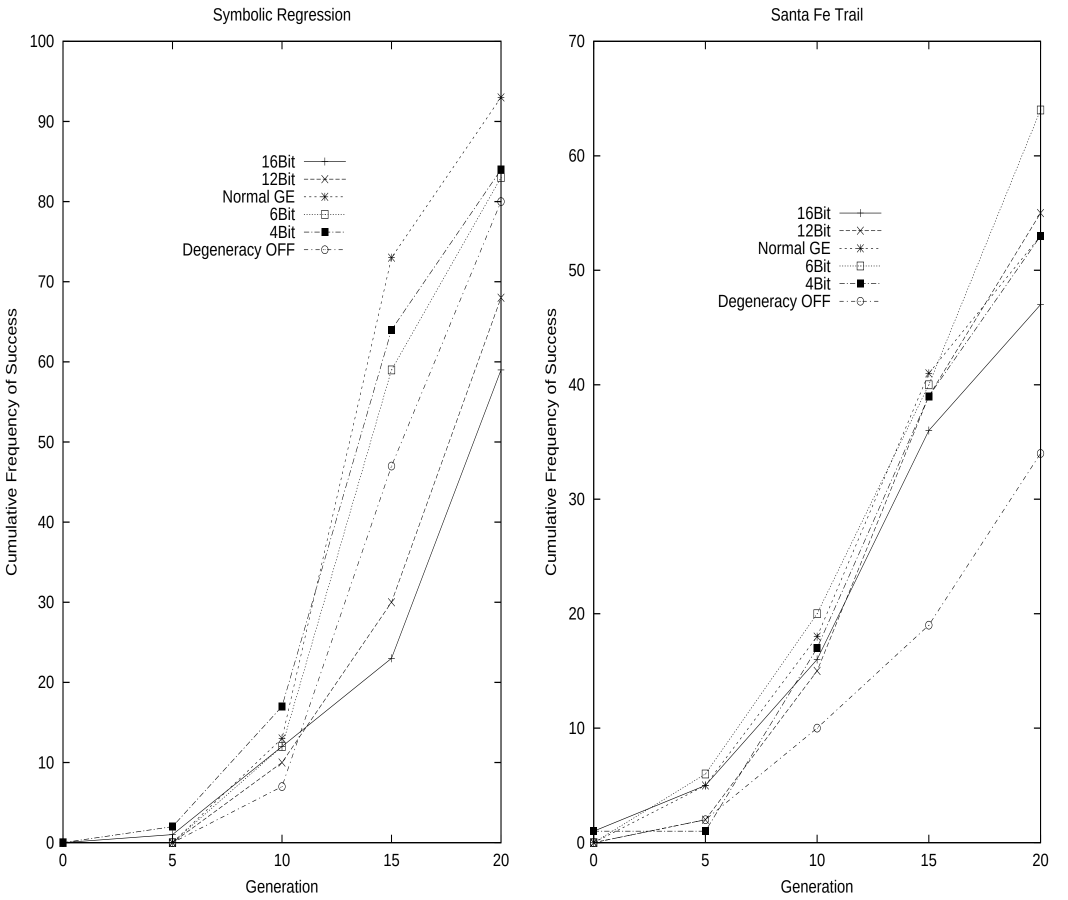


Figure 6.5: Cumulative frequencies of success for both problem domains in the presence and absence of genetic code degeneracy.

It would appear that increasing the number of bits in a codon to 16 on the symbolic regression problem resulted in slower evolutionary times. To test this we re-ran the 8, 12 and 16-bit experiments on both problems for 50 generations, and the results can be seen in Figure 6.6. The cumulative frequencies of success demonstrate the slower evolutionary times involved with the 12 and 16-bit codons, whereupon approaching 50 generations the 12 and 16-bit surpasses the 8-bit encoding on both problems.

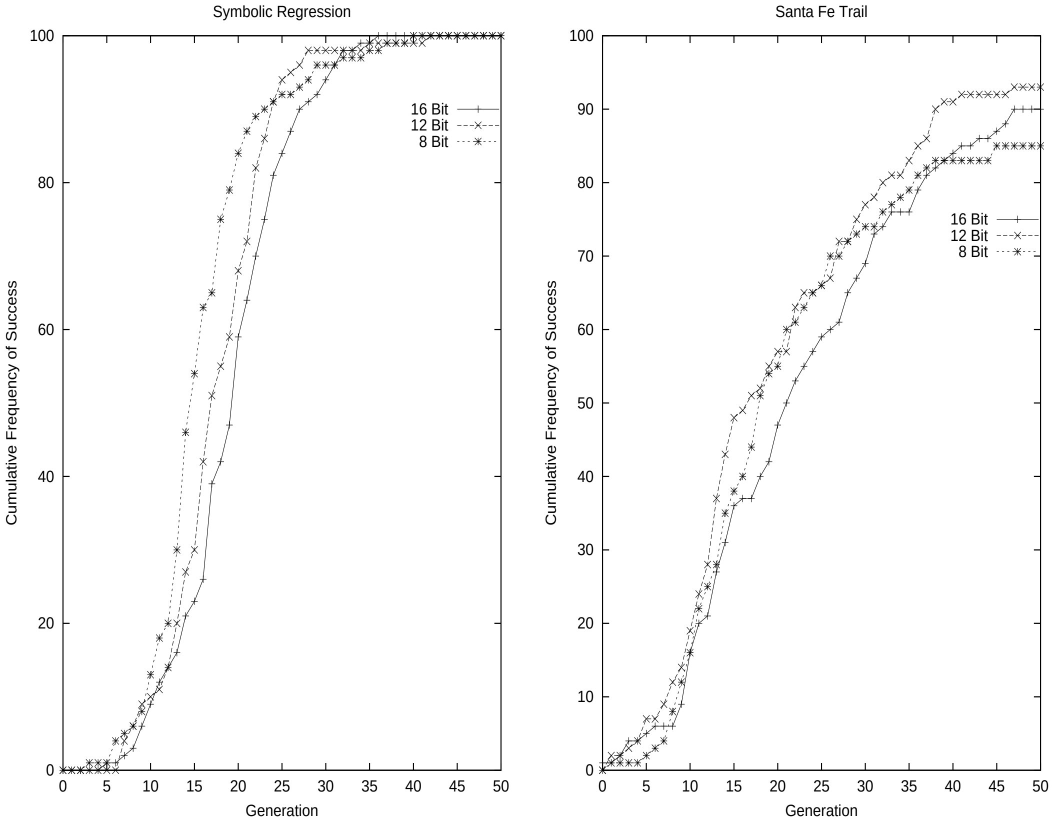


Figure 6.6: Cumulative frequencies of success for both problem domains in the presence and absence of genetic code degeneracy over 50 generations.

### Diversity Measures

The genotypic diversity measures are carried out for the 100 runs. The average of these values for each generation over all the runs can be seen in Figures 6.7 and 6.8. These graphs show that the code degeneracy has a marked effect on the mean variety measure and on the number of unique individuals. In the case where the degenerate code is removed there is a clear decrease in the mean variety and in the number of unique individuals at each generation. This suggests that the degenerate code plays a role in maintaining genetic diversity within the population.

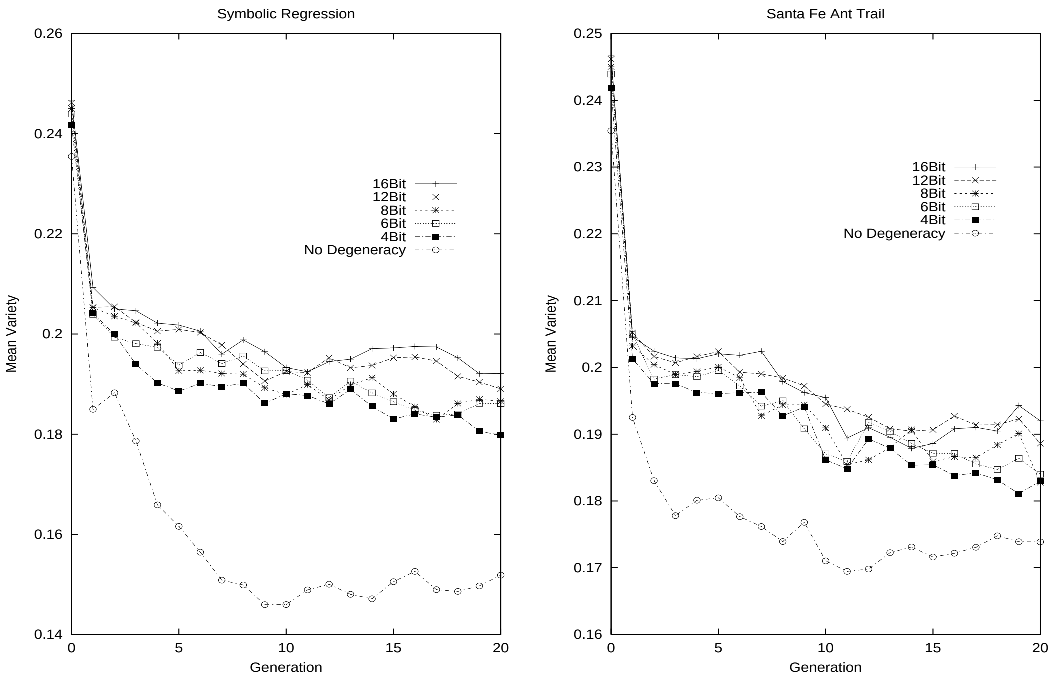


Figure 6.7: The figure shows the genetic code degeneracy and mean variety on symbolic regression and Santa Fe trail problems.

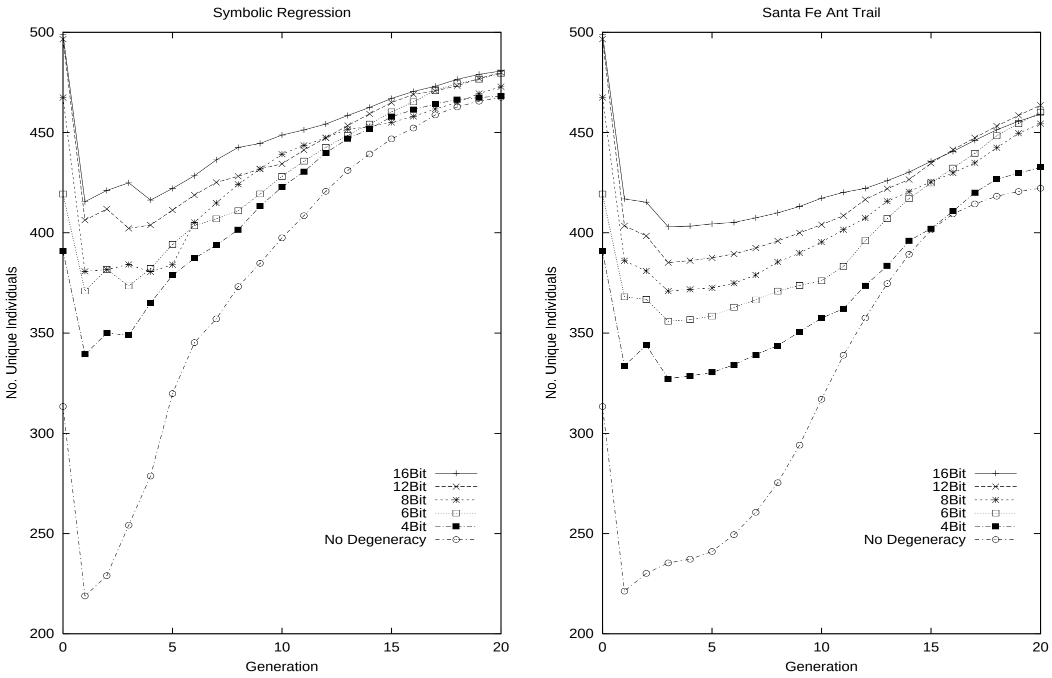


Figure 6.8: The figure shows genetic code degeneracy and unique individuals (for actual genome) on both problem domains.

### 6.3.2 Discussion

Based on the two measures used to give an indication of the genotypic diversity in a population, the results show that degeneracy in the genetic code has a beneficial effect on genotypic diversity in the population.

The fact that GE is capable of maintaining genetic diversity throughout a run could prove to be a very useful feature if the system was to be applied to problems that required an adaptive property, such as dynamic problem domains.

As well as the benefits noted for genetic diversity, the cumulative frequencies of success (Figure 6.5) show that the degenerate code improves the performance of the system in terms of its success rate at finding solutions on the more difficult Santa Fe trail problem. The results suggest that the adoption of degeneracy in GE should have positive effects on the systems performance but depending on the amount of degeneracy incorporated slower evolutionary times may result. As such, it may be possible to obtain an optimal degree of degeneracy for the problem being examined. For the problem domains examined here, with a maximum number of choices of four productions rules for any one non-terminal in the grammar, a codon of 6 to 8-bits appears appropriate (i.e. an average of 32 genetic codes representing any one production rule).

To summarise, the degenerate code maintains genetic diversity, and improves the performance of the system in terms of the success rate of runs. As the adoption of 8-bit codons produces consistent results across the problems analysed here, we recommend the use of this codon size. It must be noted, however, that we do not expect this codon size to be the optimal across all problems, indeed it may not be the optimal for the problem domains examined here.

As such, an investigation into more appropriate codon sizes may be beneficial.

Results in [Miller & Thomson, 2000] also show an improvement in performance on the Santa Fe trail problem in the presence of large redundancy in the genotype representation.

## 6.4 Removal of Wrapping and Degeneracy

A further set of experiments was conducted to determine the effects of removing both wrapping and degeneracy. Over 100 runs, on both problem domains, this set examines the cumulative frequency of success, genome lengths, and the genetic diversity measures.

### 6.4.1 Results

Figure 6.9 shows the cumulative frequency of success for the case where both wrapping and degeneracy are removed, a comparison to when both are present as in standard GE, the cases where only wrapping is switched off, and degeneracy alone is removed.

We can clearly see a degradation of performance, in terms of the success rates of finding solutions, in these graphs, with the worst performance occurring when both wrapping and degeneracy are removed.

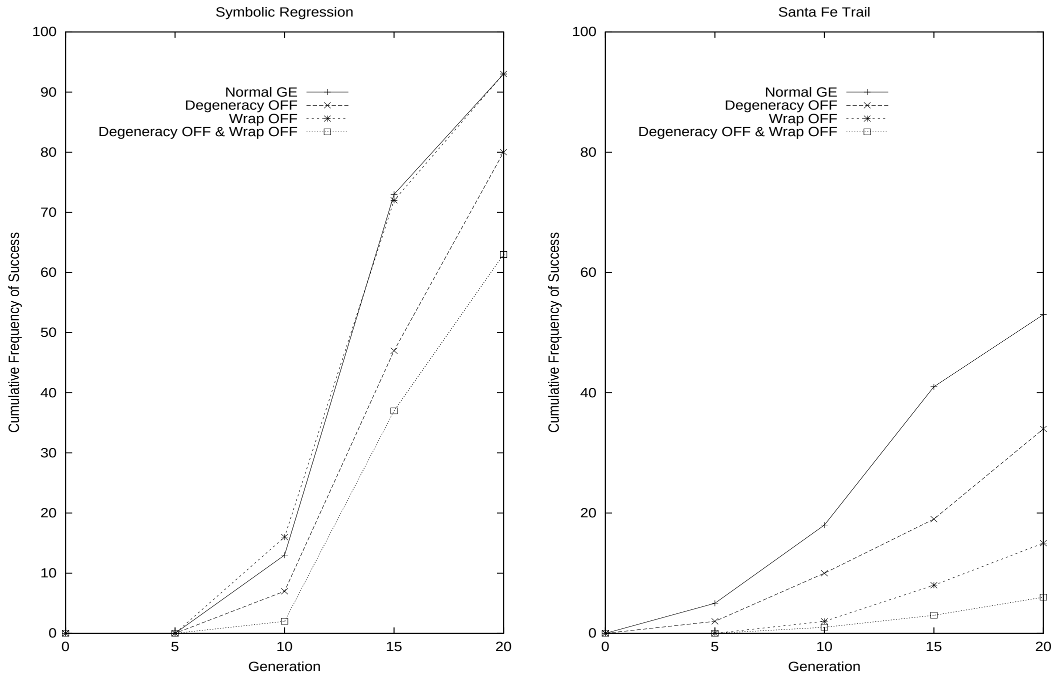


Figure 6.9: A comparison of the cumulative frequency of success on the Santa Fe trail and symbolic regression with the removal of wrapping and degeneracy, against the cases where both are present, and where each is removed individually.

Examining the genome lengths (Figure 6.10), in the case of the symbolic regression problem, we see an increase in the actual lengths as we remove each feature, until we get the largest increases when both wrapping and degeneracy are removed. On the Santa Fe trail genome lengths are considerably shorter than for symbolic regression with the trend not being as clear in this case. With wrapping switched off, and when both wrapping and degeneracy are off, we see a slight decrease in average genome lengths in the order of 5% after 20 generations. When compared with the symbolic regression results, differences in the order of 62% occur when both wrapping and degeneracy are switched off. As genome lengths are not having a detrimental affect with the

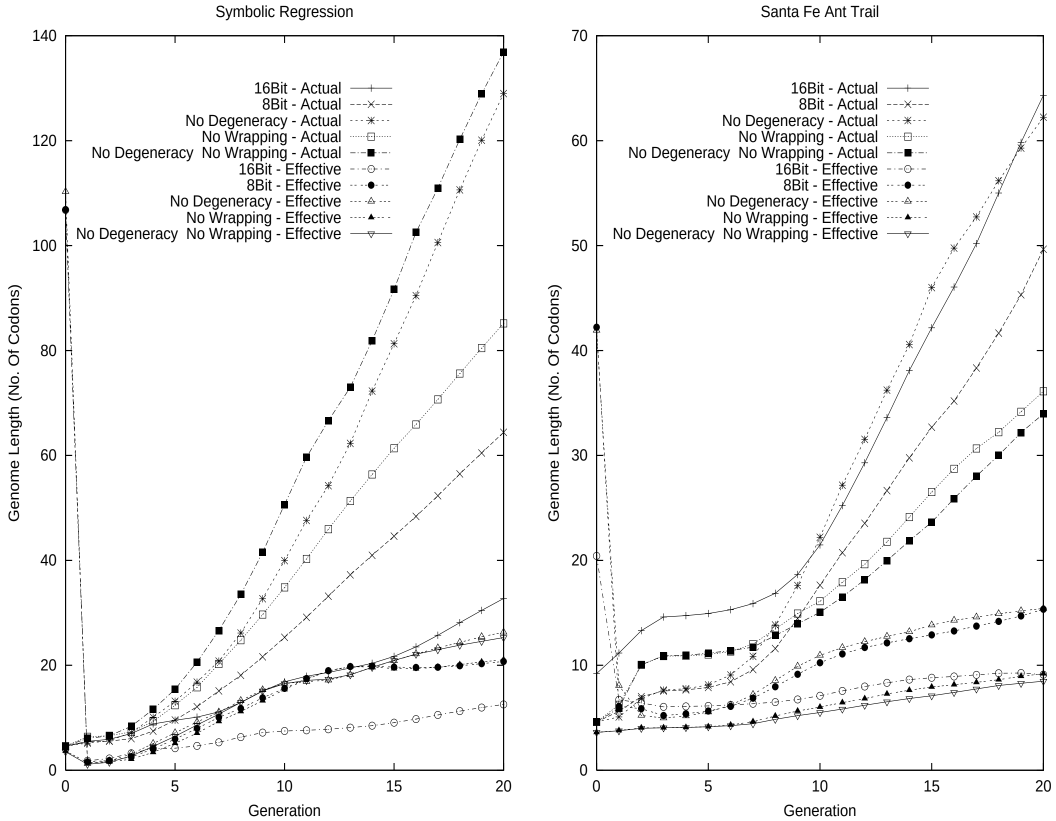


Figure 6.10: A comparison of genome lengths on both problems, with the removal of wrapping and degeneracy, against the cases where both are present, and where each is removed individually.

presence of wrapping and degeneracy on the Santa Fe trail, and are clearly beneficial in the case of symbolic regression, their continued use by GE is justified.

With respect to the genetic diversity measures we would not expect to see a further degradation in diversity with the absence of both wrapping and degeneracy, and this is illustrated in Figures 6.11 and 6.12.

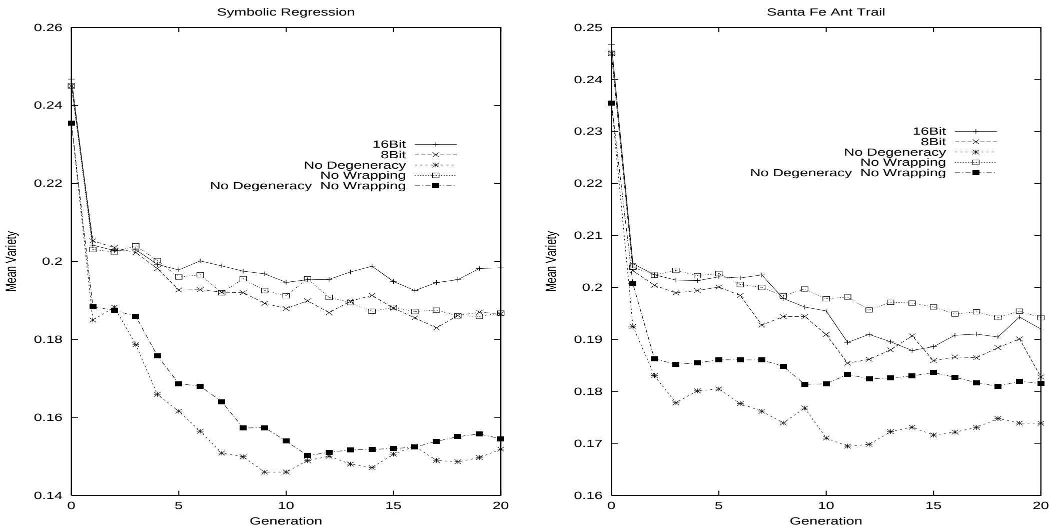


Figure 6.11: A comparison of the mean variety measure on both problems, with the removal of wrapping and degeneracy, against the cases where both are present, and where each is removed individually.

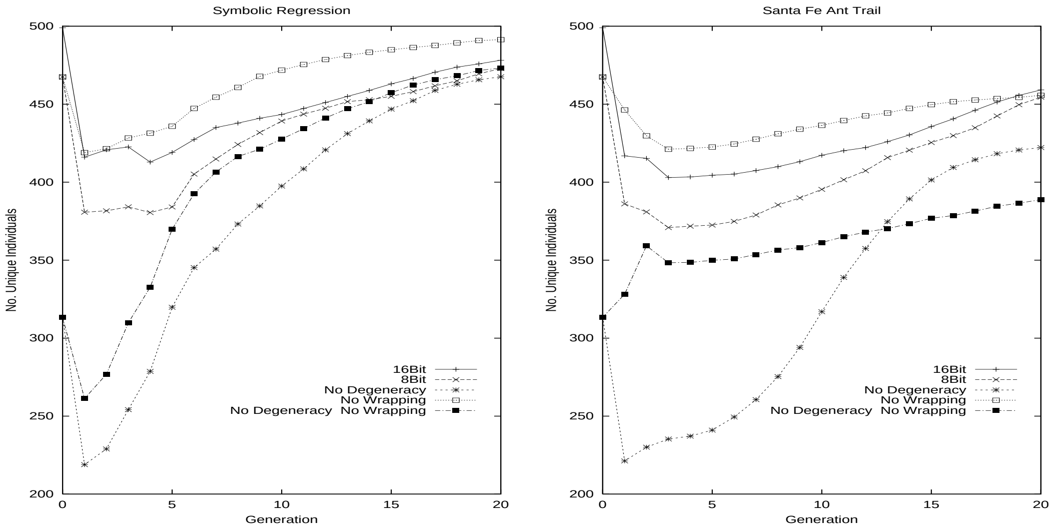


Figure 6.12: A comparison of the number of unique individuals on both problems, with the removal of wrapping and degeneracy, against the cases where both are present, and where each is removed individually.

## 6.5 Mutation Rates

A final set of experiments was conducted to investigate the possibility that an implicit change in mutation rates could be responsible for the performance effects observed with the degeneracy experiments. By removing degeneracy we reduce the number of bits in a codon, and as such we effectively reduce the mutation rates involved due to the bit mutation operator adopted. The mutation operator acts by testing each bit locus of a genome and mutating each with a pre-specified probability.

We conduct experiments whereby the mutation rate for the standard GE implementation with 8-bit codons is decreased to 0.001 from 0.01, and where the mutation rate in the case of the removal of degeneracy is increased proportionately to the new smaller codon size, to 0.0333. These results can then be compared to the standard and smaller codon size results with the usual mutation rate of 0.01.

### 6.5.1 Results

Figure 6.13 illustrates the results for these experiments on the cumulative frequency of success for both problem domains. For symbolic regression, when degeneracy is removed and the mutation rate is increased, we see a further degradation in performance. Clearly, the increased mutation rate has a disruptive effect on the evolutionary search due to the lack of degeneracy. On the Santa Fe trail problem, the cumulative frequencies are generally insensitive to the mutation rate adopted for the parameters examined here.

The results demonstrate that the degeneracy effects observed are largely as a result of the degenerate genetic code employed by GE and not an artefact of a mutation rate change.

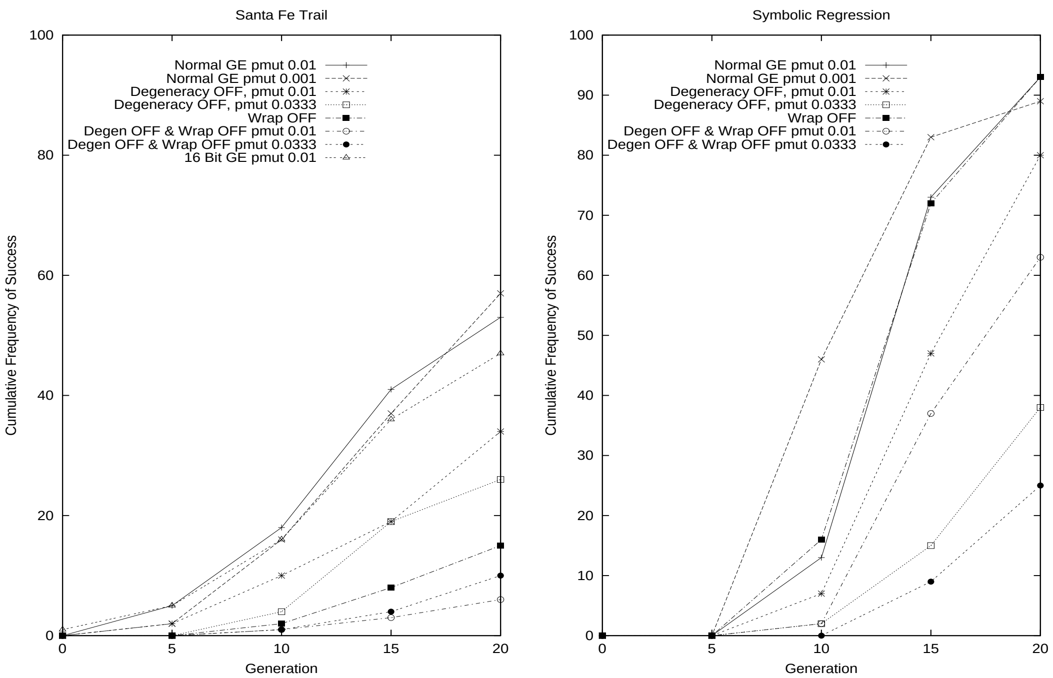


Figure 6.13: A cumulative frequency of success on the Santa Fe trail and symbolic regression with scaled mutation rates.

## 6.6 Conclusions

The effects of the wrapping operator and the degenerate genetic code have been investigated in GE. We have demonstrated the utility of the wrapping operator. We have also observed that wrapping can result in the compression of genotypic lengths. On the Santa Fe trail problem it was observed that wrapping clearly improved the success rate of GE. With respect to the degenerate genetic code we have illustrated the many benefits that this feature confers on GE, i.e. an improvement in success rate on both problem domains examined, and maintenance in genetic diversity of the population, when a degenerate genetic code is used. When both features were removed from GE, we subsequently observed a further degradation in the system's performance. Clearly, both wrapping and the degenerate genetic code play an important role in GE, at worst they have no ill effects on any of the problems examined, and at best they can improve performance. In the next chapter we go on to analyse crossover in GE.

# Chapter 7

# Crossover in Grammatical Evolution

## 7.1 Introduction

While crossover is generally accepted as an explorative operator in string based G.A.s [Goldberg, 1989], the benefit or otherwise of employing crossover in tree based Genetic Programming is often disputed. Work such as [Collins, 1992] went as far as to dismiss GP as a biological search method due to its use of trees, while [Angelin, 1997] presented results which suggested that crossover in GP can provide little benefit over randomly generating subtrees.

GE utilises linear genomes and, as with GP systems, has come under fire for its seemingly destructive crossover operator, a simple one-point crossover inspired from GA's. In this chapter we address crossover in GE, seeking answers to the question of how destructive our one-point crossover operator is, and to establish if the system could benefit from a biologically inspired crossover.

By default, GE employs a standard GA variable length, one-point crossover operator as follows: (i) Two crossover points are selected at random, one on each individual (ii) The segments on the right hand side of each individual are then swapped.

Other researchers have proposed a number of novel crossover operators [Francone et.al., 1999] [Langdon, 1999] [Langdon, 2000]. Langdon and Francone et. al. derived different homologous crossover operators, the former on tree based GP and the latter on linear structures, being introduced to improve exploration [Francone et.al., 1999] [Langdon, 1999].

Both of these works exploit the idea of a homologous crossover operation, which draws inspiration from the molecular biological crossover process. The principle being exploited is the fact that in nature the entities swapping genetic material only swap fragments that belong to the same position and are of similar size. This, it has been proposed, results in more productive crossover events. Indeed, results from both Langdon and Francone et.al. provide evidence in support of this claim.

The homologous crossover operator applied to linear genomes in [Francone et.al., 1999] is called *Sticky Crossover* and operates by swapping instructions at the same locus. It makes no attempt to swap functionally equivalent code segments. In the tree-based homologous crossover [Langdon, 2000] the crossover point on the first parent is selected as normal in GP. The crossover point on the second parent is determined by taking into account the size of the subtree created from the first crossover point, and selecting the subtree in the second parent that is closest to it. Closeness is measured by looking at the tree shapes, and at the distance between the two crossover points and the root nodes of the individuals.

A consequence of these conservative crossover operators is a reduction in the bloat phenomenon, which is due, at least in part, to the fact that these new operators are less destructive. The production of increasingly longer genomes is, therefore, deemed unnecessary. Bloat is a phenomenon whereby the sizes of individuals in a GP population increase dramatically over the duration of a run, largely due to redundant code. Suggestions have been made that destructive crossover events could be responsible for bloat; bloat arises as a mechanism to prevent destructive crossover events occurring by acting as buffering regions in which crossover can occur without harming functionality.

As with many non-binary representations, it is often not clear how much useful genetic material is being exchanged during crossover, and thus not clear how much exploration is actually taking place.

## 7.2 Homologous Crossover

This section serves to initiate our investigation of crossover in GE, and proposes a new form of operator inspired by molecular biology and the novel homologous crossover operators designed for GP. We then compare the standard GE one point crossover to two different versions of this homologous crossover, as well as two alternative forms of a two-point crossover operator.

The standard GE homologous crossover proceeds as follows:

1. During the mapping process a history of the rules selected in the grammar are stored for each individual.

2. The histories of the two individuals to crossover are aligned.

3. Each history is read sequentially from the left while the rules selected are identical for both individuals, this region of similarity is noted.

4. The first two crossover points are selected to be at the boundary of the region of similarity, these points are the same on both individuals.

5. The two second crossover points are then selected randomly from the regions of dissimilarity.

6. A two-point crossover is then performed.

An outline of this operator can be seen in Figure 7.1.

The reasoning behind this operator is the facilitation of the recombination of blocks that are in context with respect to the current state of the mapping process. In this case these blocks are of differing lengths.

The second form of the homologous crossover operator differs only in that it swaps blocks of identical size. In step 5, the two, second crossover points are at the same locus on each individual.

The two-point crossover operators employed are the standard GA two-point operator (fragments of unequal size), and one in which the size of the fragments being swapped are the same.

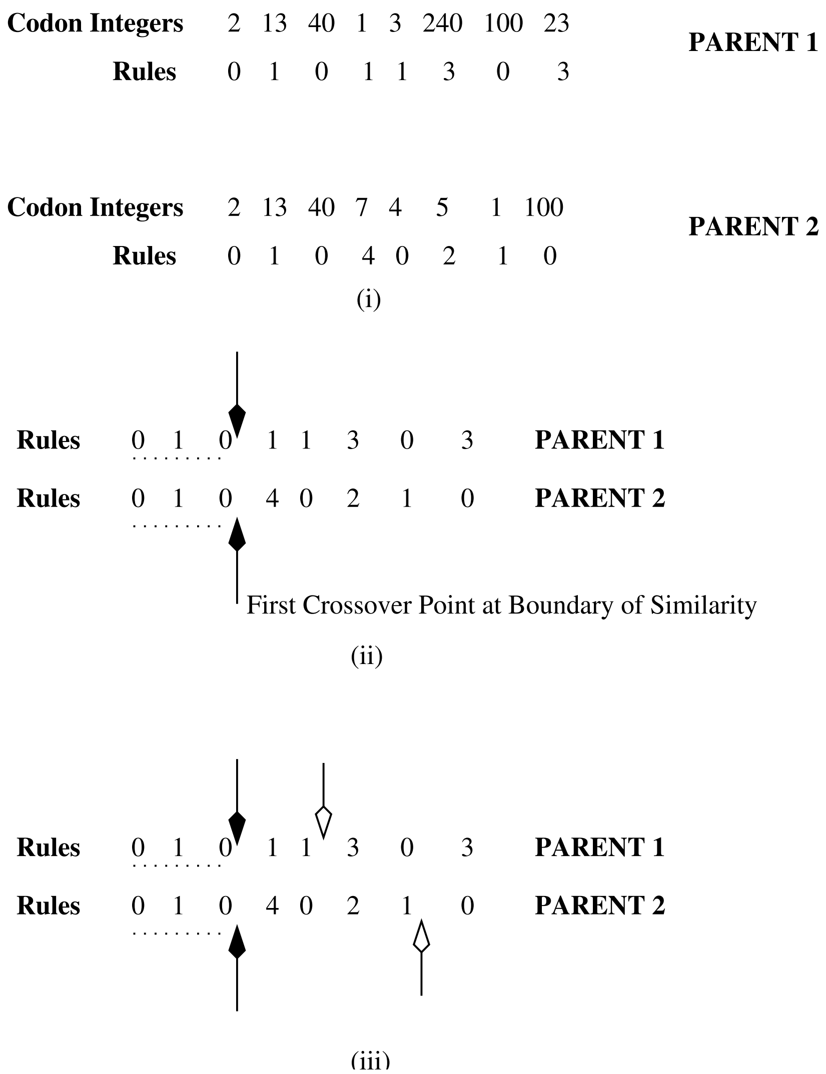


Figure 7.1: Depicted is standard GE homologous crossover. (i) Shows two parents represented as their codon integer values on top, and the corresponding rules selected during the mapping process below each integer value. (ii) The rule strings (mapping histories) are aligned, and the region of similarity noted (underlined). The first crossover points are selected at this boundary. (iii) The second crossover points are then selected after the boundary of similarity for each individual.

### 7.2.1 Experimental Approach

For each type of crossover, 100 runs were carried out on the Santa Fe trail and Symbolic regression problems, as described in Chapter 5. Performance of each operator was measured in terms of each of the following

1. Cumulative frequency of success
2. Average size of fragments being swapped at each generation
3. Ratio of the average fragment size being swapped to the average genome length
4. Ratio of crossover events resulting in successful propagation of the individual to the next generation and the total number of crossover events.

Measures (2) and (3) have been used previously in [Poli & Langdon, 1998]. Measure (4) is used to determine the productiveness of the crossover operator by looking at the number of individuals that are propogated to the next generation after having undergone crossover. Tableaus describing the parameters and terminals are given in Chapter 5.

### 7.2.2 Results

As can be seen in Figures 7.2 and 7.3, the cumulative frequencies of success clearly show that standard one and two point crossover are superior to the other operators on both problem domains. We will now describe the results for each operator under the other measures described in the previous section.

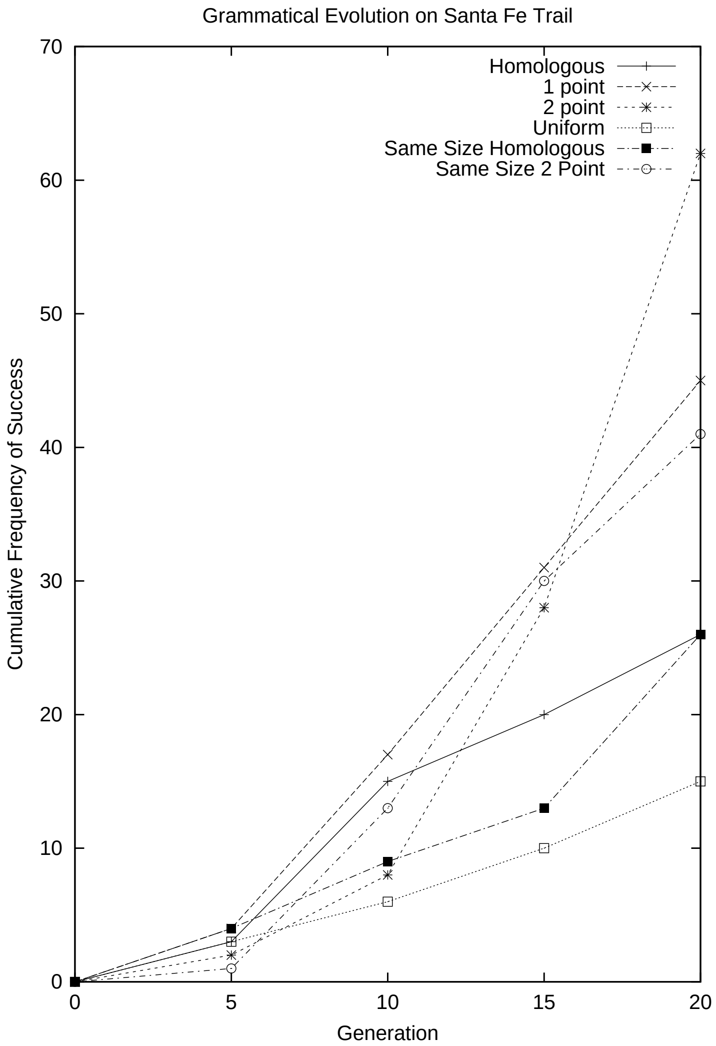


Figure 7.2: Comparison of the cumulative frequencies of success for each crossover operator on the Santa Fe ant trail problem.

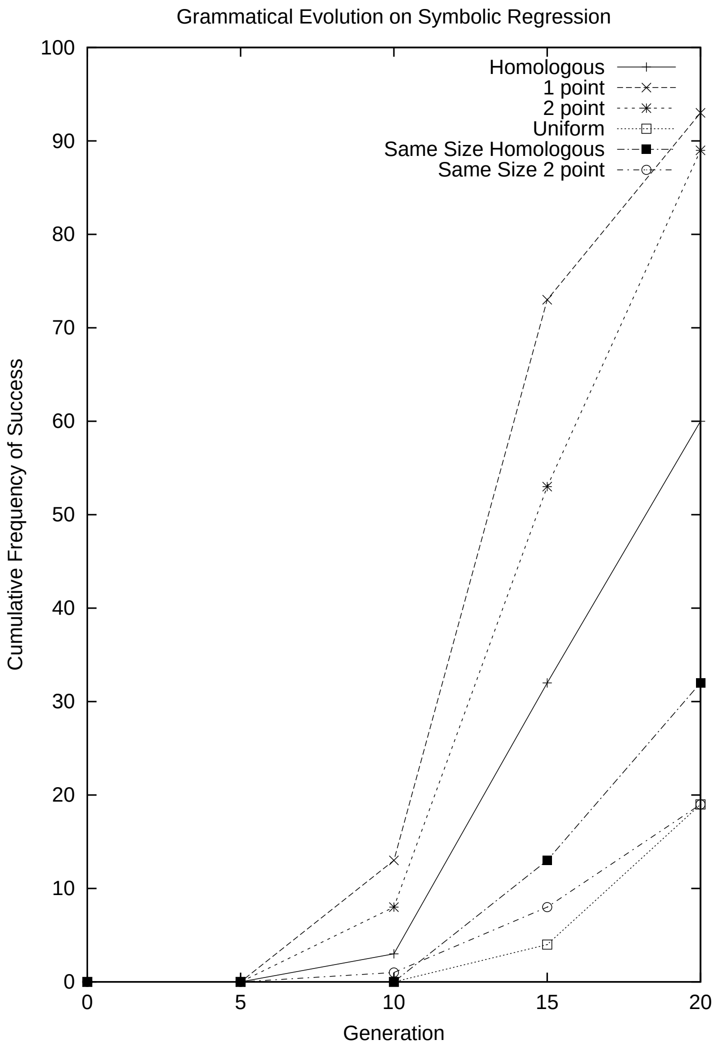


Figure 7.3: Comparison of the cumulative frequencies of success for each crossover operator on the Symbolic Regression problem.

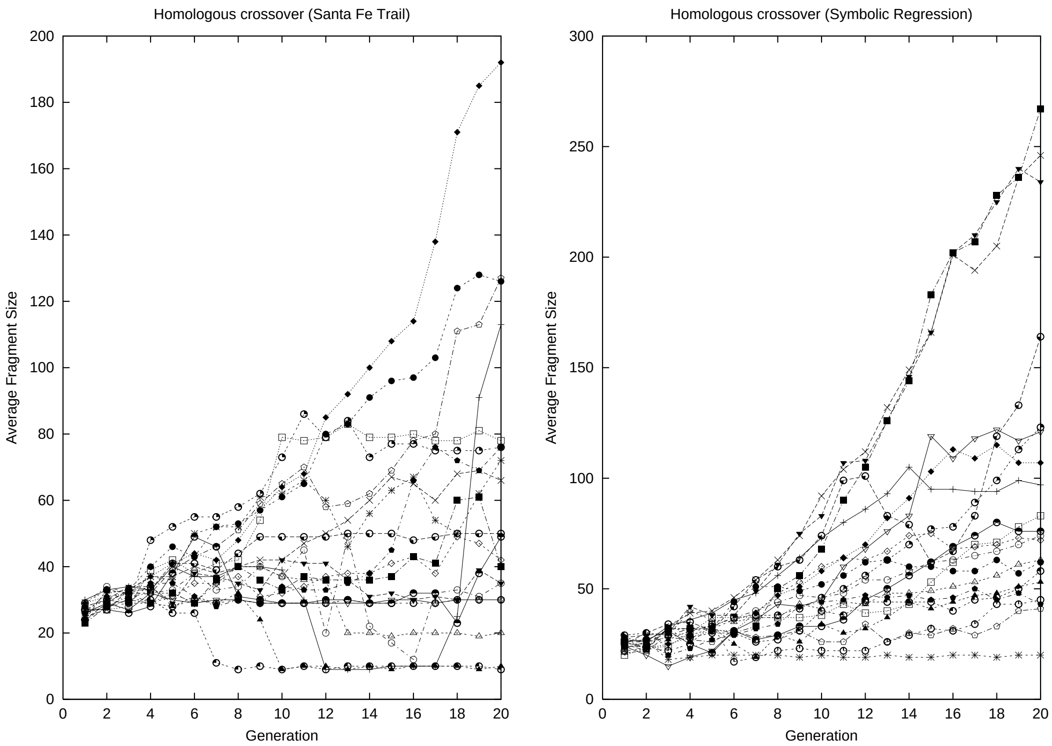


Figure 7.4: Average fragment size being swapped each generation for Homologous crossover

Figure 7.4 shows the average fragment size being swapped at each generation with homologous crossover. Data is presented for 20 separate runs and plotted in this manner because we are interested in general trends over the set of runs as opposed to the precise details for individual runs in each of these graphs.

Overall the fragment size increases as each generation passes, although the lengths of the chromosomes are also increasing. As such, it is difficult to see what is happening to the fragment size. A more useful measure of the ratio of average fragment size to the average chromosome length can be seen in Figure 7.5.

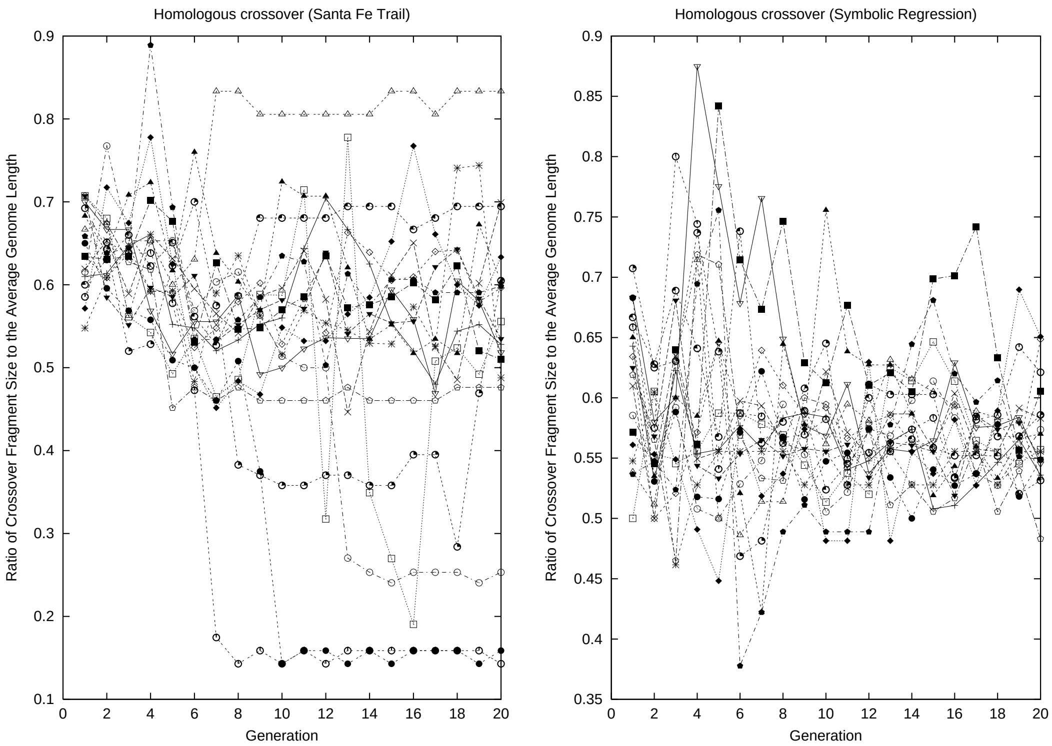


Figure 7.5: Ratio (centered around 0.6) of the average fragment size being swapped and the average chromosome length at each generation for Homologous crossover

Similar graphs for same size homologous, two point, same size two point, and one point can be seen in Figures 7.6, 7.7, 7.8, and 7.9 respectively.

For each of the homologous crossover operators and the one point operator, on average 50% of the genetic material is being exchanged during these events. It would appear that these operators are acting as global search operators.

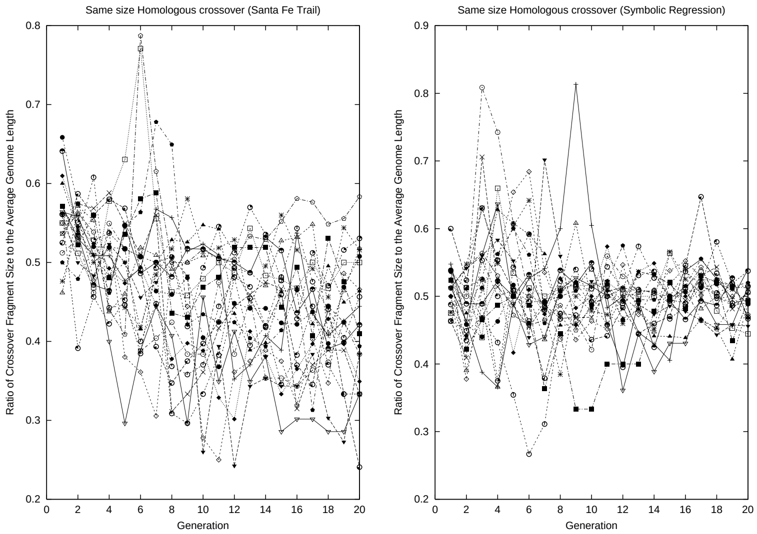


Figure 7.6: Ratio (centered around 0.5) of the average fragment size being swapped and the average chromosome length at each generation for Same size Homologous crossover

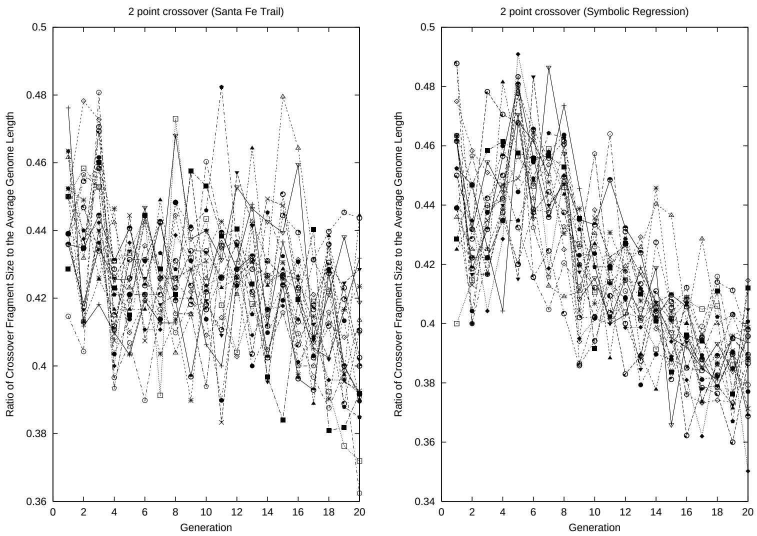


Figure 7.7: Ratio (centered around 0.43) of the average fragment size being swapped and the average chromosome length at each generation for two point crossover

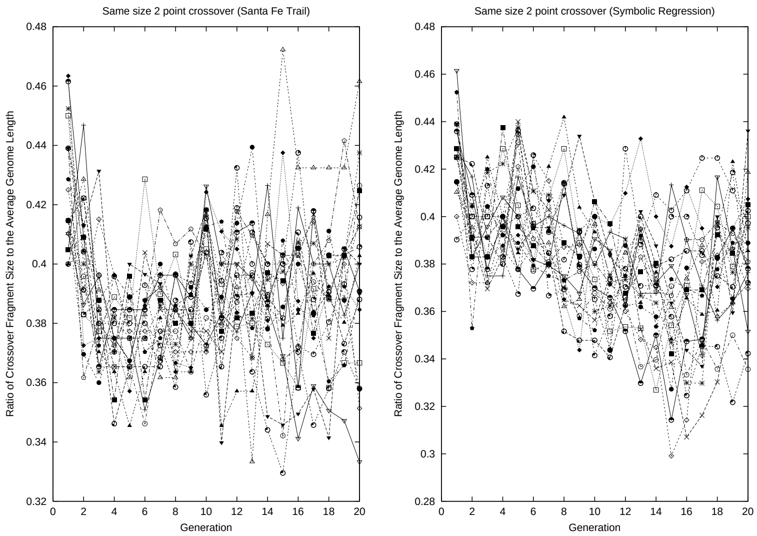


Figure 7.8: Ratio (centered around 0.4) of the average fragment size being swapped and the average chromosome length at each generation for same size two point crossover

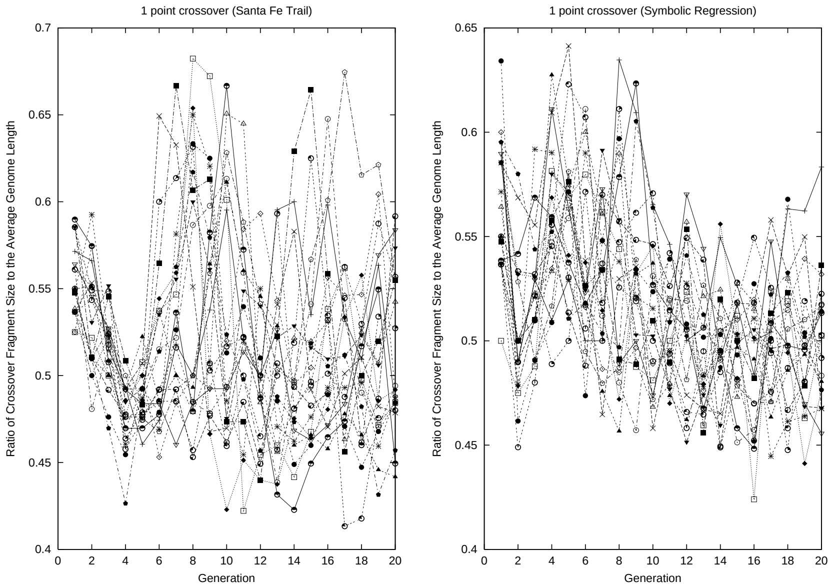


Figure 7.9: Ratio (centered around 0.5) of the average fragment size being swapped and the average chromosome length at each generation for one point crossover

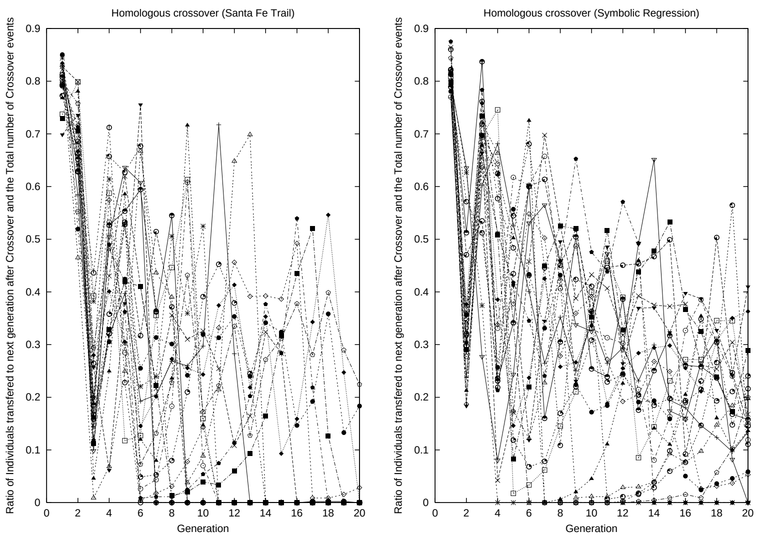


Figure 7.10: Ratio of the number of individuals undergoing Homologous crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation.

Figure 7.10 shows the ratio of individuals undergoing crossover that have been successfully propagated to the next generation and the total number of crossover events that have occurred. The results of this measure for all other operators can be seen in Figures 7.11, 7.12, 7.13, and 7.14. For both homologous operators there is no obvious trend to the data, indeed, the transmission of individuals as a result of homologous crossover to the next generation would appear to be erratic. Both of the two point operators and the one point crossover each have clearer trends. The two-point operator does appear to be less successful on this measure. On the Santa Fe trail, propagation is more erratic than on the symbolic regression problem. We propose this is as a result of the dependency on the use of the wrapping operator that would have an effect of making crossover more disruptive.

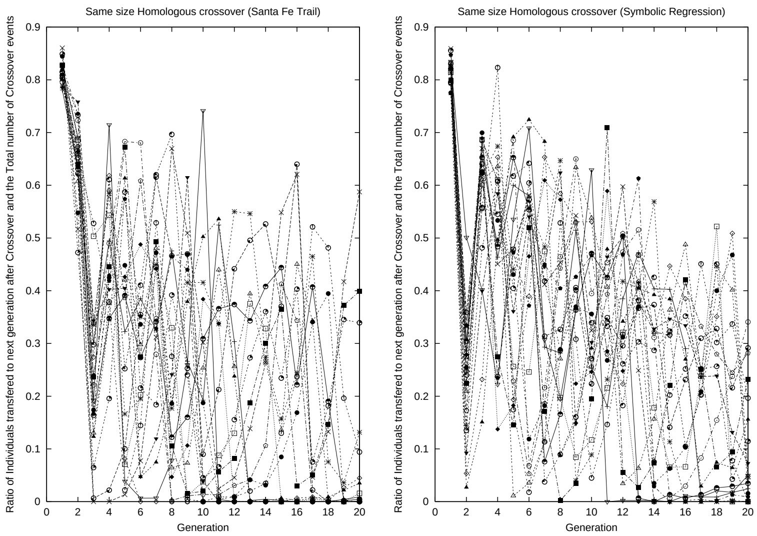


Figure 7.11: Ratio of the number of individuals undergoing Same size Homologous crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation.

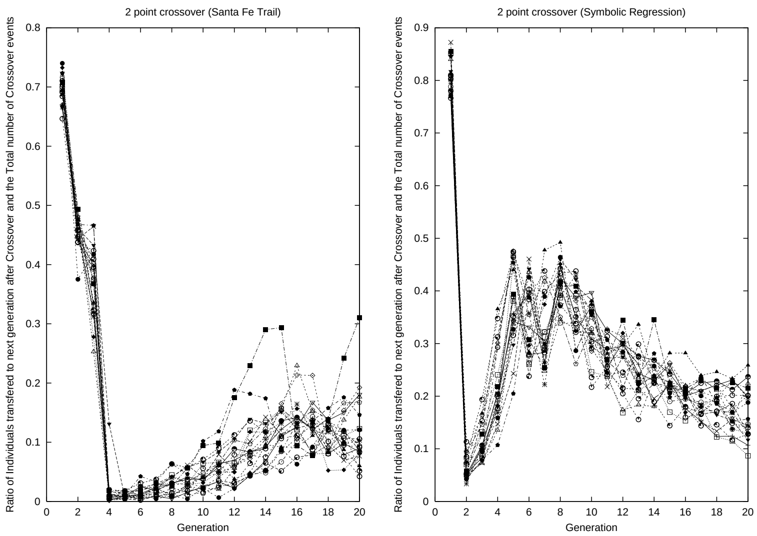


Figure 7.12: Ratio of the number of individuals undergoing two point crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation.

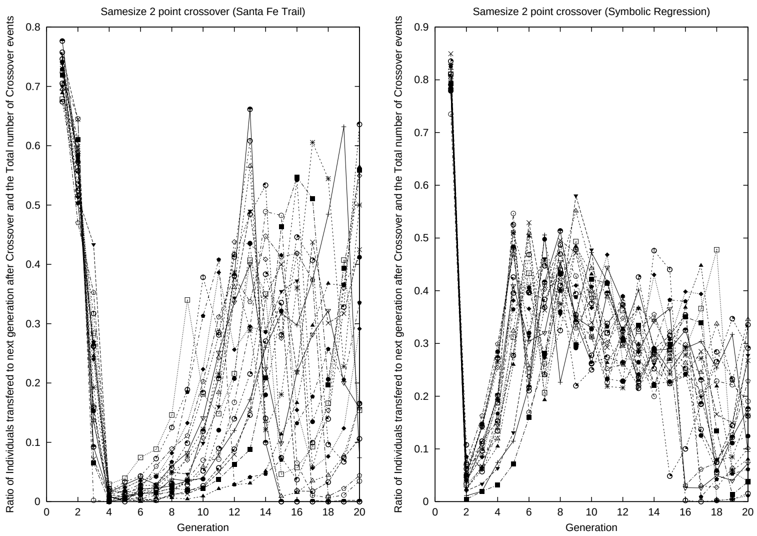


Figure 7.13: Ratio of the number of individuals undergoing same size two point crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation.

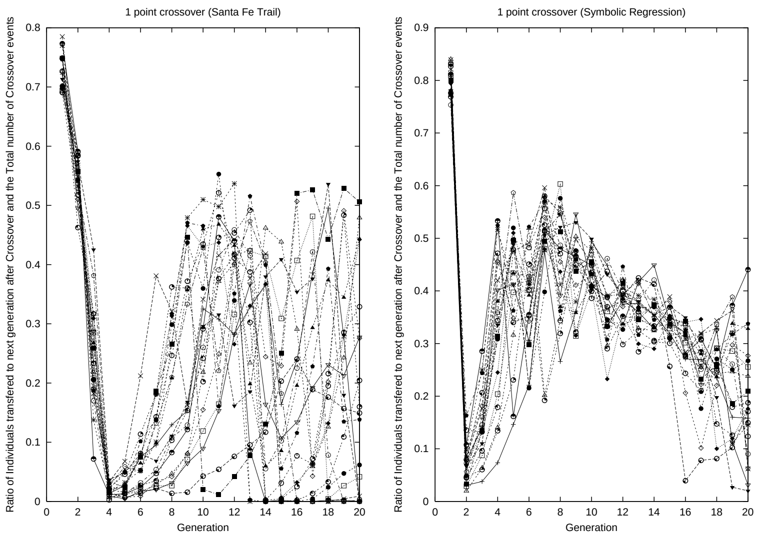


Figure 7.14: Ratio of the number of individuals undergoing one point crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation.

### 7.2.3 Discussion

Examining Figures 7.15 and 7.16, we can see the average over 20 runs of the ratio of individuals undergoing crossover that are propagated to the next generation to the total number of crossover events for each generation, and the ratio of the average crossover fragment size to the average chromosome length, respectively. In terms of the ratio of individuals being propagated to the next generation having undergone crossover, we can see that for one point crossover in the case of the Symbolic Regression problem the rate of transfer remains relatively constant throughout the run around the value 0.35. A similar trend can be seen for one point crossover in the case of the Santa Fe trail problem although there is a slight deterioration as runs progress. Looking at the ratio of individuals transfered to the total number of crossover events for individual runs (Figures 7.10, 7.11, 7.12, 7.13 and 7.14), we can see in all cases that in the first few generations this value is extremely high. These results for homologous crossover are not as consistent as those for one point, and, directly compared, appear erratic. They show that the effort required to carry out our version of homologous crossover, and the occasional peak it achieves in terms of individual transfer to each generation, are outweighed by the consistent results produced by the much simpler one point operator.

In general though, we can see the utility of one point crossover by virtue of the fact that it produces individuals that are capable of being propagated to successive generations, given the steady state replacement strategy.

Looking at the ratio of the average crossover fragment size to the average chromosome length (Figure 7.16), in the case of one-point crossover we get a relatively smooth line around the 0.5 mark for both problems. Similar trends are observed for both types of two-point crossover; however, these are

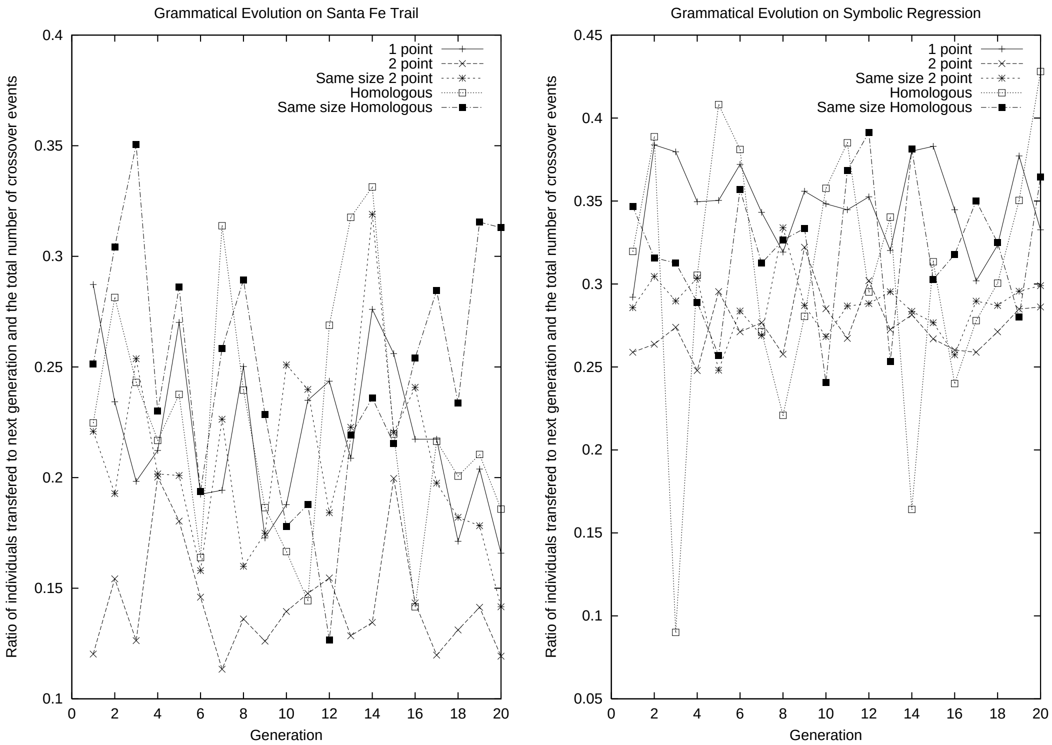


Figure 7.15: Ratio of the number of individuals undergoing crossover that have been propagated to the next generation and the total number of crossover events occurring in that generation averaged over 20 runs.

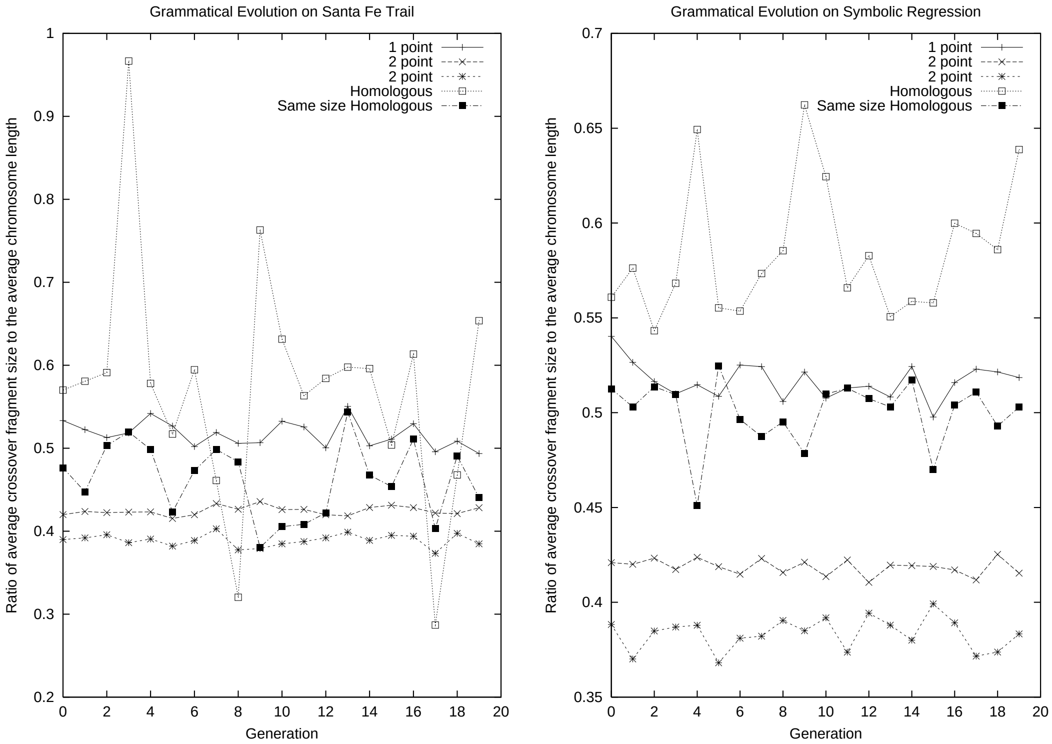


Figure 7.16: Ratio of the average fragment size being swapped and the average chromosome length at each generation averaged over 20 runs.

localised to lower values. It can also be seen that the homologous operators are more erratic, although on the symbolic regression it has a consistently higher value than all other operators. The experimental evidence shows us that crossover results in individuals that are being propagated to the next generation, and that the size ratio of these fragments to the actual genome length is consistent throughout the runs. This is in direct contrast to the results obtained in [Poli & Langdon, 1998] which showed a drop off for all crossover operators except, naturally, for the uniform operator. These results showed how the operators examined in this case start off as global search operators but change rapidly to local search operators.

In light of the data obtained, namely the transfer of individuals having undergone crossover to the next generation, it is reasonable to suspect that within GE individuals there exist useful building blocks that are being recombined to produce better performing individuals. With respect to the homologous operator described in [Langdon, 1999] for tree based GP, a lot of effort is required to carry out this operation, whereas in GE, we get a simple, efficient crossover operator with less effort, that exchanges half the material on average.

## 7.3 Headless Chicken

We now continue the analysis of crossover in GE by firstly turning off crossover, and secondly, by exchanging random blocks in a headless chicken-type crossover [Angelin, 1997]. Two forms of headless chicken crossover were described for trees, *strong headless chicken crossover* (SHCC) being the most similar to the one adopted here. SHCC operated by creating a random tree for each of the parents selected for crossover, followed by standard subtree crossover in GP. The modified parent tree (as opposed to the modified randomly generated tree) being returned. The other form of headless chicken crossover (*weak headless chicken crossover*) returned randomly either the modified parent or the modified random tree.

The objective of this set of experiments is to confirm observations made pertaining to the effectiveness of crossover in the previous set of experiments.

### 7.3.1 Experimental Approach

We wish to examine the strength of crossover in GE using a two-pronged approach. Firstly, the probability of crossover will be set to zero to switch off the operator. Secondly, we will replace the standard one point crossover adopted with a headless chicken-type crossover. The headless chicken operator we adopt selects the fragments to crossover, and replaces them with a randomly generated bit string of the same lengths.

For each experiment 50 runs were carried out on the Santa Fe ant trail and a symbolic regression problem. Performance was ascertained by the cumulative frequency of success.

### 7.3.2 Results

Results for the experiments can be seen in Figure 7.17. These graphs clearly demonstrate the damaging effects of the headless chicken crossover, and in the case when crossover is switched off. On the symbolic regression problem GE fails to find solutions in both of these cases, while on the Santa Fe ant trail, the system's success rate falls off dramatically.

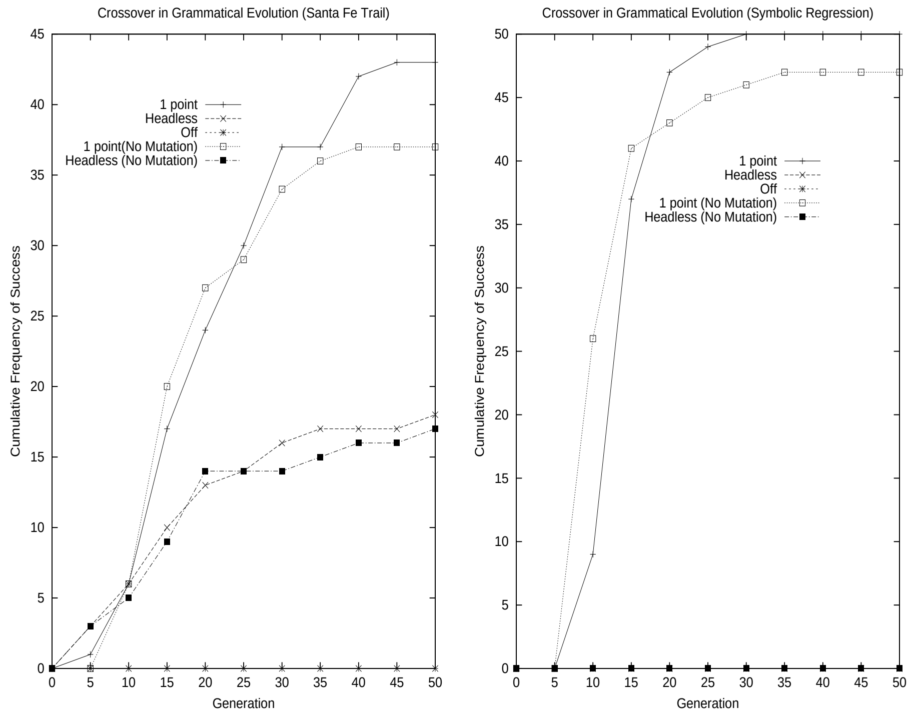


Figure 7.17: A comparison of GE's performance on the Santa Fe ant trail can be seen on the left. The graph clearly demonstrates the damaging effects of the headless chicken crossover, and in the case when crossover is switched off. A comparison of GE's performance on the symbolic regression problem can be seen on the right. When the headless chicken crossover is used the system fails to find solutions, as is also the case when crossover is switched off.

These results clearly demonstrate the power of GE's one point crossover as an operator that successfully exploits an exchange of useful blocks on the problems examined. It also demonstrates that the one point crossover operator is essential to the effective operation of the system.

Experiments were also conducted where the mutation operator was turned off in the cases of both one-point and the homologous crossover operators. The results, see Figure 7.17, demonstrate that, when mutation is turned off in the presence of one-point crossover, there is a decrease in performance when compared to the case where mutation is present. These observations apply to both problem domains examined, and, as such, suggest that mutation plays a beneficial role alongside the successful one-point crossover operator.

### 7.3.3 Discussion

The question arises then as to why GE's one point crossover operator is so productive. If we look at the effect the operator plays on a parse tree representation of the programs undergoing crossover, we begin to see more clearly the mechanism of this operator and its search properties.

When mapping a string to an individual, GE always works with the left most non-terminal. Thus, if one were to look at the individual's corresponding parse tree, one would see that the tree is constructed in a pre-order fashion. Furthermore, if the individual is over-specified, that is, has codons left over, they form a *tail*, which is effectively a stack of codons, as illustrated in Figure 7.18.

If, during a crossover event, one tried to map the first half of the remaining strings, the result, not surprisingly, would usually be an incomplete tree. However, the tree would not be incomplete in the same manner as one taken from the middle of a GP crossover event.

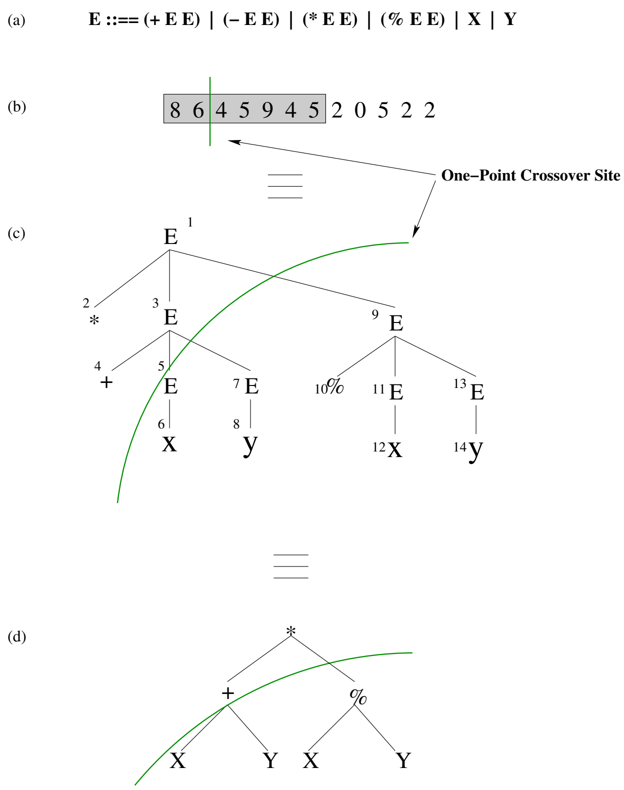


Figure 7.18: The ripple effect of one-point crossover illustrated using an example GE individual represented as a string of codon integer values (b) and its equivalent derivation (c) and parse trees (d). The codon integer values in (b) represent the rule number to be selected from the grammar outlined in (a), with the part shaded gray corresponding to the values used to produce the trees in (c) and (d), the remaining integers are an intron. Figure 7.19 shows the resulting spine with ripple sites and tails.

The pre-order nature of the mapping is such that the result is similar to that of Figure 7.18 and Figure 7.19. That is, the tree is left with a *spine* and several *ripple sites* from which one or more sub-trees, dubbed *ripple trees*, are removed. This crossover behaviour, which is an inherent property of GE, was first noticed in [Keijzer et.al., 2001] where it was termed *ripple crossover*.

Each of the ripple trees is effectively dismantled and returned to the stack of codons in the individual's tail. Crossover then involves individuals swapping tails so that, when evaluating the offspring, the ripple sites on the spine will be filled using codons from the other parent.

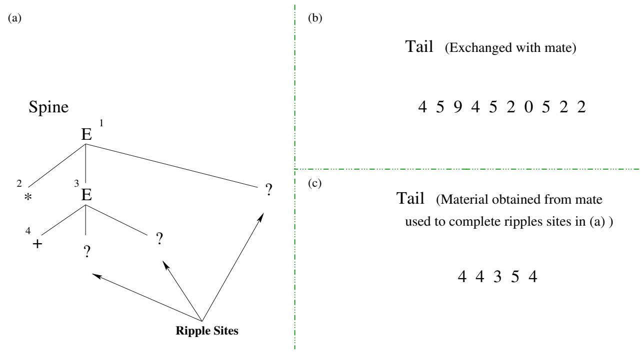


Figure 7.19: Illustrated are the spine and the resulting ripple sites (a) and tails (b)(c) produced as a consequence of the one-point crossover in Figure 7.18

There is no guarantee that the tail from the other parent will be of the same length, or even that it is used in a similar place on the other spine. This means that a codon that represented which choice to make could suddenly be expected to make a choice from a completely different non-terminal, possibly with a different number of choices. Fortunately, GE evaluates codons *in context*, that is, the exact meaning of a codon is determined by those codons that immediately precede it. Thus, we can say that GE codons have *intrinsic polymorphism*, as they can be used in any part of the grammar; furthermore, if the meaning of one codon changes, the change cascades, or "ripples" through all the rest of the codons. This means that a group of codons that coded a particular sub-tree on one spine can code an entirely different sub-tree when employed by another spine. The power of intrinsic polymorphism can even reach between the ripple trees, in that if one no longer needs all its codons, they are passed to the next ripple tree and, conversely, if it requires more codons, it can obtain them from its neighbouring ripple tree.

## 7.4 Conclusions

We began this chapter with an investigation into a new homologous crossover operator designed for GE, to discover that it was no better than the standard one-point crossover originally adopted.

Further analysis, using a headless chicken-type operator, and by switching off crossover altogether, revealed the power this one-point operator brought to GE.

In the writing of this chapter, the value of linear chromosomes in general, and the GE system in particular, became quite clear to us. Ripple crossover occurs effectively for free in a linear system, because of the pre-order nature of tree construction, and results in, on average, 50% of the material being exchanged during a crossover event. Furthermore, the phenomenon of *intrinsic polymorphism* demonstrates the utility of context sensitive genes (groups of codons), that is, genes that can change their behaviour depending on the manner in which they are used. Rather elegantly, although the genes are polymorphic, they will always return to their initial state if used in the same manner again.

# Chapter 8

# Conclusions & Future Work

## 8.1 Summary

We have presented Grammatical Evolution (GE), a system that is capable of evolving computer programs in an arbitrary language. This ability is achieved through the adoption of a genotype-phenotype distinction, that generates the output program by the application of production rules from a grammar. The grammar is used to specify the output language and its syntax, therefore allowing program generation in any langauge. A number of additional features inspired by Molecular Biology are achieved. These include: a separation of the search and solution spaces that allow the EA search engine to be a plug-in component of the system, facilitating the exploitation of advances in EA's by GE; a many-to-one, degenerate genetic code that allows the exploitation of neutral evolution to enhance the search efficiency of the EA; and the use of a wrapping operator, that allows the reuse of genetic material during a genotype-phenotype mapping process.

Following from a review of Automatic Programming in which the focus of attention was drawn towards Evolutionary Automatic Programming, this thesis examined a number of potential advantages that could be achieved through a genotype-phenotype distinction in Chapter 3, and later showed evidence of their existence in GE in Chapters 4 and 6. The ability of this approach to evolve computer programs using proof of concept problems was demonstrated in Chapter 5, with the performance of GE shown to be on a par with, and in some cases superior to, Genetic Programming.

An analysis of GE was conducted in Chapters 6 and 7 and a number of interesting results were obtained. It was found that the novel wrapping operator was useful in the completion of successful genotype-phenotype mappings, and that some compression of the genotype was observed. The degenerate genetic code was found to have desirable features, such as maintenance of genetic diversity within the population, an improvement in the success rate of runs, and a preservation of the functionality of evolving programs whilst the evolutionary search is still being conducted. A detailed analysis of crossover in GE determined that the one-point crossover adopted has a dramatic positive effect on the performance of the system. This effect, we suggest, is due to the global search that is conducted by virtue of its mechanism that becomes apparent when examined in terms of parse trees, by swapping on average half of the genetic material.

## 8.2 Future Work

Various questions are posed as a result of this thesis, and many avenues of investigation have yet to be explored. The following is a list of possible lines of research that could be profitable to GE and to the Evolutionary Automatic Programming community at large.

**The Grammar**

The grammar adopted by GE is a context-free grammar that is used to ensure the syntactic correctness of the evolving programs. The next step in the development of GE would be to investigate the use of attribute and logic grammars. These context-sensitive grammars extend the expressive power of a context-free grammar by taking into account the context of the current non-terminal in the code as it is being generated. These context-sensitive grammars would allow the incorporation of information on types, facilitating the evolution of multi-typed programs.

The grammar could also be used as a mechanism to incorporate an equivalent of the automatically defined functions of Genetic Programming. Some initial investigations with encouraging results have been conducted in [O'Neill & Ryan, 2000]. This approach will come into its own when it is possible to dynamically change the grammar used during the mapping process of GE. This will enable dynamic automatic function definition where the number and type of functions no longer has to be pre-determined, but rather would be evolved.

No effort has been made during the course of this thesis to conduct an analysis of different grammars and their effects on the performance of GE. During the experiments we have conducted, very simple, tailored grammars were adopted. It is possible to include the entire grammar for a language such as C into this system, or indeed to use different variations on the grammars that we have adopted. As noted in [Whigham, 1996b] it is possible to use the grammar to incorporate bias into the evolutionary search. It is therefore, necessary to discover what effects different grammars would have on GE.

#### Transcription Model

The manner in which the genome is read and converted into codon integer values is a very simplistic transcription process model. Many potential avenues of investigation are possible with this model. One example would include the investigation of different wrapping strategies, for example, using different reading frames after a wrap event. More complex models such as the operon model (as described in Chapter 3) might be implemented.

The incorporation of introns into GE is another area that might have benefits. Conflicting reports exist on the benefit or otherwise of introns in evolutionary computation, and, as such, it would be interesting to test their effects on GE.

#### Translation Model

Throughout the course of this work the mapping function that comprises the translation model of GE has remained static. The mapping function

$$Rule = (Codon\ integer\ value)$$

$$MOD$$

$$(Number\ of\ rules\ for\ the\ current\ non-terminal)$$

has proved to be a powerful component of GE being responsible for the intrinsic polymorphism properties that can arise during crossover events (see Chapter 7). It would be interesting to explore variations on the translation model with the use of different mapping functions.

In conjunction with the grammar it would also be possible to test the effects of evolving the genetic code on-line.

### The Evolutionary Algorithm Engine

For the sake of simplicity we have adopted a simple GA for the proof of concept problems and system analysis. As the scalability of the simple GA to harder problems has been shown to be poor, it would be useful to study the benefits of the so-called competent GAs (GAs with improved scaling characteristics) on the scalability of GE.

Other modifications to the evolutionary algorithm engine could include concepts such as diploidy and polygenic inheritance, particularly if the system was to be applied to dynamic problem domains.

### The Initialisation Strategy

The creation of generation zero has been a purely random approach up until now. As alternative intialisation strategies are found to be of benefit to GP, the investigation of similar strategies in GE may yield efficiency gains.

### Neutral Evolution

No analysis has been conducted on the evolutionary dynamics during this work, but we have speculated on the exploitation of neutral evolution by GE. In order to determine if we are benefiting from neutral evolutionary processes, further investigations must be conducted. We are particularly interested in adopting visualisation techniques that are being used to give insights into the dynamics of other evolutionary algorithms.

**Application Areas**

One of the major steps forward will be the application of GE to real-world problems. We demonstrated success with GE's application to some standard benchmark problems and a real-world caching algorithm problem in Chapter 5. The flexibility of GE in its ability to produce programs in an arbitrary language will facilitate its application to many diverse problem domains. According to the No Free Lunch theorem, GE won't be optimal at all problems, but it would be interesting to determine for what classes of problems this type of Evolutionary Automatic Programming system is useful.

# Appendix A

## Publications

Parts of this thesis have appeared in the various publications detailed below.

### Journals

- M. O'Neill & C. Ryan. Grammatical Evolution. IEEE Transactions on Evolutionary Computation, 2001.

### Conference Papers

- M. Keijzer, C. Ryan, M. O'Neill, M. Cattolico, & V. Babovic. Ripple Crossover in Genetic Programming. To appear in the Proceedings of EuroGP 2001.
- M. O'Neill, C. Ryan, M. Keijzer & M. Cattolico. Crossover in Grammatical Evolution: The Search Continues. To appear in the Proceedings of EuroGP 2001.
- M. O'Neill & C. Ryan. Incorporating Gene Expression Models into Evolutionary Algorithms. In *Proceedings of the 2000 Genetic and*

*Evolutionary Computation Conference Workshop Program*, Las Vegas, pages 167-172.

- M. O'Neill & C. Ryan. Grammar based function definition in Grammatical Evolution. In *Proceedings of GECCO 2000, the Genetic and Evolutionary Computation Conference*, Las Vegas, pages 485-490

- M. O'Neill & C. Ryan. Crossover in Grammatical Evolution: A Smooth Operator? In *Proceedings of the Third European Workshop on Genetic Programming 2000*, Edinburgh, Scotland, pages 149-162.

- M. O'Neill & C. Ryan. Automatic Generation of Programs with Grammatical Evolution. In *Proceedings of AICS 1999*, University College Cork, Cork, Ireland, pages 72-78.

- M. O'Neill & C. Ryan. Automatic Generation of Programs that Outperform Human Designed Algorithms. In *Proceedings of the Sixth Research Conference of the Dept. of Computer Science and Information Systems*, University of Limerick, September 1999.

- M. O'Neill & C. Ryan. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. In *Proceedings of the European Conference on Artificial Life, 1999.*

- M. O'Neill. Automatic Programming with Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program*, July 13-17, 1999, Orlando, Florida USA. San Francisco, CA: Morgan Kaufmann.

- M. O'Neill & C. Ryan. Under the Hood of Grammatical Evolution. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (eds.). *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, July 13-17, 1999, Orlando, Florida USA. San Francisco, CA: Morgan Kaufmann.

- M. O'Neill & C. Ryan. Automatic Generation of Caching Algorithms. In *Proceedings of EUROGEN 1999, Short Course on Evolutionary Algorithms in Engineering and Computer Science*, Jyvaskyla, Finland, May 30 - June 3, 1999, pages 127-134.

- M. O'Neill & C. Ryan. Evolving Multi-Line Compilable C Code. In *Proceedings of the Second European Workshop on Genetic Programming 1999*, pages 83-92.

- M. O'Neill & C. Ryan. Automatic Generation of High Level Functions using Evolutionary Algorithms. In *Proceedings of SCASE 1999, Soft Computing and Software Engineering Workshop*, University of Limerick, Ireland.

- M. O'Neill & C. Ryan. Grammatical Evolution: A Steady State approach. In *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms 1998*, pages 419-423.

- M. O'Neill & C. Ryan. Grammatical Evolution. In *Proceedings of the Fifth Research Conference of the Dept. of Computer Science and Information Systems*, University of Limerick, September 1998.

- M. O'Neill & C. Ryan. Grammatical Evolution: A Steady State approach. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, July 22-25,1998. Madison, WI: Omni Press.

- C. Ryan, J.J. Collins, & M. O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings First European Workshop on Genetic Programming 1998.* Lecture Notes in Computer Science 1391.

# Bibliography

[Angelin, 1997] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Banzhaf, 1994] Wolfgang Banzhaf. Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, pages 322–332, Jerusalem, 9-14 October 1994. Springer-Verlag.

[Banzhaf et.al., 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.

[Barnett, 1997] L. Barnett. Evolutionary dynamics on fitness landscapes with neutrality. Master's thesis, School of Cognitive Sciences, University of East Sussex, Brighton, 1997.

[Barreau, 2000] G. Barreau. *The Evolutionary Consequences of Redundancy in Natural and Artificial Genetic Codes.* PhD thesis, University of Sussex, 2000.

[Blume & Eigenmann, 1992] W. Blume and R. Eigenmann. Performance analysis pf parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[Chomsky, 1956] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[Collins, 1992] R. Collins. *Studies in Artificial Life.* PhD thesis, University of California, Los Angeles, 1992.

[Collins & Ryan, 1999] J. J. Collins and Conor Ryan. Non-stationary function optimization using polygenic inheritance. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, page 781, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Cramer, 1985] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, July 1985.

[Crick, 1966] F.H.C. Crick. Codon-anticodon pairing; the wobble hypothesis. *Journal of Molecular Biology*, 19:548–555, 1966.

[Cructhfield & van Nimwegen, 1999a] Crutchfield, James P. van Nimwegen, Erik. Metastable evolutionary dymnamics: Crossing fitness barriers or escaping via neutral paths? Technical Report 99-07-041, Santa Fe Institute, 1999.

[Cructhfield & van Nimwegen,1999b] Crutchfield, James P. van Nimwegen, Erik. Neutral evolution of mutational robustness. Technical Report 99-03-021, Santa Fe Institute, 1999.

[Darwin, 1859] Charles Darwin. *On the Origins of the Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* 1859.

[De Jong, 1999] K. De Jong. *Evolutionary Algorithms in Engineering and Computer Science*, chapter 3. Evolutionary Computation: Recent Developments and Open Issues, pages 43–54. Wiley, 1999.

[Eigen et.al., 1989] M. Eigen, J. Mc Caskill, and P. Schuster. The molecular quasispecies. *Adv. Chem. Phys.*, 75:149–263, 1989.

[Engelhardt, 1998] Engelhardt, Robin. *Emergent Percolating Nets in Evolution.* PhD thesis, Center for Chaos and Turbulence Studies, Univeristy of Copenhagen, 1998.

[Engelhardt & Newman, 1998] Engelhardt, Robin. Newman, M.E.J. Effects of neutral selection on the evolution of molecular species. *Proc. R. Soc. London B*, 1998.

[Flanagan, 1992] K. Flanagan et. al. Bach: Byu address collection hardware. Technical Report TR-A150-92.1., Electrical and Computer Engineering Department, Brigham Young University, 1992.

[Fogel et.al., 1966] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley, 1966.

[Foster, 1991] Ian Foster. Automatic generation of self-scheduling programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):68–78, January 1991.

[Francone et.al., 1996] Frank D. Francone, Peter Nordin, and Wolfgang Banzhaf. Benchmarking the generalization capabilities of a compiling genetic programming system using sparse data sets. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 72–80, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Francone et.al., 1999] Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous crossover in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1021–1026, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Freeman, 1998] Jennifer J. Freeman. A linear representation for GP using context free grammars. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 72–77, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Friedberg, 1958] R.M. Friedberg. A learning machine: Part 1. *IBM J. Research and Development*, Vol. 2:1:2–13, 1958.

[Friedberg et.al., 1959] R.M. Friedberg, B. Dunham, and J.H. North. A learning machine: Part 2. *IBM J. Research and Development*, pages 282–287, 1959.

[Geyer-Schulz, 1997] Andreas Geyer-Schulz. The next 700 programming languages for genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 128–136, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Goldberg, 1987] D. Goldberg. Nonstationary function optimisation with dominance and diploidy. In *Second International Conference on Genetic Algorithms*, 1987.

[Goldberg, 1989] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1989.

[Goldberg et.al, 1989] D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[Gruau, 1994] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[Harik, 1999] G.R. Harik. Linkage learning via probabilistic modeling in the ecga. Technical Report IlliGAL Report No. 99010, University of Illinois at Urbana-Champaign, 1999.

[Harik & Goldberg] G.R. Harik and D.E. Goldberg. *Foundations of Genetic Algorithms 4*, chapter Learning Linkage, pages 247–262.

[Holland, 1980] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1980.

[Hollstein, 1971] R.B. Hollstein. *Artificial genetic adaptation in computer control systems*. PhD thesis, University of Michigan, 1971.

[Horner, 1996] H. Horner. A C++ class library for genetic programming: The vienna university of economics genetic programming kernel. Vienna University of Economics, 1996.

[Huynen, 1995] Huynen, Martijn. Exploring phenotype space through neutral evolution. Technical Report 95-10-100, Santa Fe Institute, 1995.

[Huynen et.al., 1996] Fontana, W. Huynen, M.A., Stadler, P.F. Smoothness within ruggedness: the role of neutrality in apdaptation. *Proc. Natl. Acad. Sci. (USA)*, 93:397–401, 1996.

[Jacob & Monod, 1961] F. Jacob and J. Monod. Genetic regulatory mechanisms in the synthesis of proteins. *Journal of Molecular Biology*, 3:318–356, 1961.

[Kargupta, 1997] Hillol Kargupta. Relation learning in gene expression: Introns, variable length representation, and all that. Position paper at the Workshop on Exploring Non-coding Segments and Genetics-based Encodings at ICGA-97, 21 July 1997.

[Kargupta, 1998] H. Kargupta. Revisiting the gemga: Scalable evolutionary optimization through linkage learning. In *Proc. of IEEE International Conference on Evolutionary Computation*, pages 603–608. IEEE Press, 1998.

[Kargupta & Sarkar, 1999] Hillol Kargupta and Kakali Sarkar. Function induction, gene expression, and evolutionary representation construction. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 313–320, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Keijzer et.al., 2001] Keijzer M., Ryan C., O'Neill M., Cattolico M., and Babovic V. Ripple crossover in genetic programming. In *Proceedings of EuroGP 2001*, 2001.

[Keller & Banzhaf, 1996] Robert E. Keller and Wolfgang Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Keller & Banzhaf, 1999] Robert E. Keller and Wolfgang Banzhaf. The evolution of genetic code in genetic programming. In Wolfgang

Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1077–1082, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Kennedy, 1998] Paul J. Kennedy. *Simulation of the Evolution of Single Celled Organisms with Genome, Metabolism and Time-Varying Phenotype.* PhD thesis, University of Technology, Sydney, 1998.

[Kimura, 1983] Motoo Kimura. *The Neutral Theory of Molecular Evolution.* Cambridge University Press, 1983.

[Koza, 1989] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N.S. Sridharan, editor, *Proceedings of the 11th International Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, 1989.

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

[Koza, 1994] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[Koza, 2001] John R. Koza. www.genetic-programming.org: A source of information about the field of genetic programming. http://www.genetic-programming.org, March 2001.

[Koza et.al., 1999] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[Langdon, 1998] William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[Langdon, 1999] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[Langdon, 2000] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming And Evolvable Machines*, 1(1/2):95–119, April 2000.

[Langdon & Poli, 1998] W. B. Langdon and R. Poli. Why ants are hard. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Lewin, 1999] B. Lewin. *Genes VII*. Oxford University Press, 1999.

[Lovely, 1992] R. Lovely. Loft: A tool for automatic parallelisation of Fortran programs. In M. Valero, E. Onate, M. Jane, J. L. Larriba, and B. Suarez, editors, *Parallel Computing and Transputer Applications*, pages 277–286, Amsterdam, The Netherlands, 1992. IOS Press.

[Miller & Thomson, 2000] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Muggleton, 1992] Stephen Muggleton, editor. *Inductive logic programming.* London : Academic, 1992.

[Naur, 1963] P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[Ng & Wong, 1995] K. Ng and K. Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimisation. In *Fifth International Conference on Genetic Algorithms*, 1995.

[Nordin, 1994] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[Nordin, 1997] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[Nordin, 1998] Peter Nordin. AIMGP: A formal description. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[Nordin & Banzhaf, 1995] Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Nordin et.al., 1999] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[O'Neill & Ryan, 1999] M. O'Neill and C. Ryan. Genetic code degeneracy: Implications for grammatical evolution and beyond. In *ECAL'99: Proc. of the Fifth European Conference on Artificial Life*, Lausanne, Switzerland, 1999.

[O'Neill & Ryan, 2000] M. O'Neill and C. Ryan. Grammar based function definition in grammatical evolution. In D. Whitley, D. Goldberg,

E. Cantú-Paz, L. Spector, I. Parmee, and H-G. Beyer., editors, *GECCO 2000: Proc. of the Genetic & Evolutionary Computation Conference*, pages 485–490, Las Vegas, Nevada, 2000. Morgan Kaufmann.

[Ošmera et.al., 1997] P. Ošmera, V. Kvasnička, and J. Pospíchal. Genetic algorithms with diploid chromosomes. In *Proceedings of Mendel '97, 3rd International Mendel Conference on Genetic Algorithms, Fuzzy Logic, Neural Networks, and Rough Sets*, pages 111–116, Brno, Czech Republic, June 1997.

[Paterson & Livesey, 1996] Norman R. Paterson and Mike Livesey. Distinguishing genotype and phenotype in genetic programming. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 141–150, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[Paterson & Livesey, 1997] Norman Paterson and Mike Livesey. Evolving caching algorithms in C by genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 262–267, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Pelikan et. al., 2000] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. Linkage problem, distribution estimation, and bayesian networks. *Evolutionary Computation*, 8(3):311–340, 2000.

[Poli & Langdon, 1998] Riccardo Poli and William B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Reidys, 1995] C.M. Reidys. *Neutral Networks of RNA Secondary-strcutures.* PhD thesis, Friedrich Schiller Universität Jena, 1995.

[Reidys et.al, 1997] C.M. Reidys, P.F. Stadler, and P. Schuster. Generic properties of combinatory maps - neutral networks of rna secondary structures. *Bull. Math. Biol.*, 59:339–337, 1997.

[Reidys et.al., 1998] C.M. Reidys, C.V. Forst, and P. Schuster. Replication and mutation on neutral networks of rna secondary structures. *Bull. Math. Biol.*, 1998. In Press, Santa Fe Institute Preprint 98-04-036.

[Rich & Waters, 1988] C. Rich and R.C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, pages 40–51, August 1988.

[RML] RML Technologies Inc. Discipulus. http://www.aimlearning.com.

[Ryan, 1996] Conor Ryan. *Reducing Premature Convergence in Evolutionary Algorithms*. PhD thesis, University College, Cork, Ireland, 2 July 1996.

[Ryan, 1999] Conor Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming*. Kluwer Academic Publishers, 1 November 1999.

[Ryan & Collins, 1998] C. Ryan and J.J. Collins. Polygenic inheritance - a haploid scheme that can outperform diploidy. In *Fifth Int. Conf. on Parallel Problem Solving from Nature*, LNCS 1498, pages 178–187, Amsterdam, September 1998. Springer.

[Ryan & Ivan, 1999] Conor Ryan and Laur Ivan. An automatice software re-engineering tool based on genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 2, pages 15–39. MIT Press, Cambridge, MA, USA, June 1999.

[Ryan & Walsh, 1997] Conor Ryan and Paul Walsh. The evolution of provable parallel programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 295–302, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Ryan et.al., 1998] C. Ryan, J.J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *EuroGP'98: Proc. of the First European Workshop on Genetic Pro-*

*gramming*, Lecture Notes in Computer Science 1391, pages 83–95, Paris, France, 1998. Springer.

[Samuel, 1959] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[Spencer, 1864] Herbert Spencer. *The Principles of Biology*, volume 1. Williams and Norgate, London and Edinburgh, 1864.

[Teller, 1994] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[Thierens, 1999] D. Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.

[Vassilev & Miller, 2000] V.K. Vassilev and J.F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *ICES 2000. Evolvable Systems: From Biology to Hardware*, pages 252–263, Edinburgh, Scotland, 2000.

[Whigham, 1995a] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995.

[Whigham, 1995b] P. A. Whigham. Inductive bias and genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, Sheffield, UK, 12-14 September 1995. IEE.

[Whigham, 1996a] P. A. Whigham. Search bias, language bias, and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Whigham, 1996b] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning.* PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, 14 October 1996.

[Wong, 1995] Man Leung Wong. *Evolutionary Program Induction Directed by Logic Grammars.* PhD thesis, Department of Computer Science and Engineering. The Chinese University of Hong Kong, 1995.

[Wong & Leung, 1994] Man Leung Wong and Kwong Sak Leung. Inductive logic programming using genetic algorithms. In J. W. Brahan and G. E. Lasker, editors, *Advances in Artificial Intelligence - Theory and Application II*, pages 119–124. I.I.A.S., Ontario, Canada, 1994.

[Wong & Leung, 1995] Man Leung Wong and Kwong Sak Leung. Applying logic grammars to induce sub-functions in genetic programming. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 737–740, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[Wong & Leung, 1997] Man Leung Wong and Kwong Sak Leung. Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180, summer 1997.

[Wong & Leung, 2000] Man Leung Wong and Kwong Sak Leung. *Data Mining Using Grammar Based Genetic Programming and Applications*, volume 3 of *Genetic Programming*. Kluwer Academic Publishers, January 2000.

[Yu & Bentley, 1998] Tina Yu and Peter Bentley. Methods to evolve legal phenotypes. In Agoston E. Eiben, Thomas Back, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Fifth International Conference on Parallel Problem Solving from Nature*, volume 1498 of *LNCS*, pages 280–291, Amsterdam, 27-30 September 1998. Springer.