Social Programming

Investigations in Grammatical Swarm

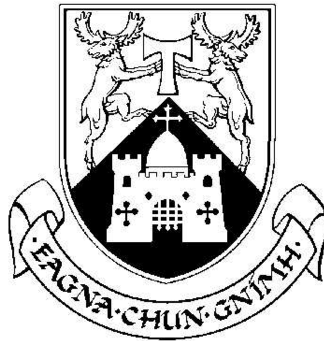
by

Finbar Leahy

B.Sc.

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

M.Sc Software Engineering



Faculty of Computer Science & Information Systems

UNIVERSITY *of* LIMERICK
OLLSCOIL LUIMNIGH

Autumn 2005

Colophon

Print Version: No.2 – 16 October 2005
Document Typeset: $\LaTeX 2_\varepsilon$.
Figures: XFig.
Graphs: GNUPLOT.
Source Code Implemented in ANSI C/C++.
Compiler: g++ (GCC) 3.4.2 under (Red Hat 3.4.2-6.fc3)
Interpreter: S-lang version 1.4.9
Libraries: Grammatical Evolution - libGE version 1.23Beta2
Derived Paper: *Grammatical Swarm: A Variable-Length Particle Swarm Algorithm*.

Abstract

This study details a series of investigations examining a recently introduced form of automatic programming called *Social Programming*. The Grammatical Swarm algorithm is a form of *Social Programming* as it uses Particle Swarm Optimisation, a social swarm algorithm, for the automatic construction of computer programs for the optimisation of continuous, non-linear problems.

An investigation into the performance effects of two different quality Pseudo-Random Number Generators (PRNG) on the Grammatical Swarm algorithm was examined. The results demonstrate that the choice of PRNG does, in fact, have a small effect of the performance of the Grammatical Swarm, with the more sophisticated PRNG producing better results on two of the four problems analysed.

An investigations was conducted into the effects of increasing the size of the particle representations of the Grammatical Swarm algorithm, such that the hard-length vector constraint of all particles in the swarm was doubled from 100 to 200. The results demonstrated that this leads to a significant gain in performance.

This thesis also introduces a new variable-length form of the Grammatical Swarm algorithm. Thus, this can be considered a proof of concept study. It examines the possibility of constructing programs using a particles representations which are variable in length and it is referred to as the Variable-Length Grammatical

Swarm. This newly developed algorithm extends earlier work on the fixed-length incarnation of Grammatical Swarm, where each individual represents choices of program construction rules, where these rules are specified using a Backus-Naur Form grammar. The results demonstrate that is possible to successfully generate programs programs using a variable-length Particle Swarm Algorithm. This investigation also examines the performance effects of increasing the initialisation size of the variable-length particles. The results demonstrate that the performance of the Variable-Length Grammatical Swarm can be increased by doubling the potential size of the particle representations.

Furthermore, the *evolution of size* in the particle representations is examined. This investigation was conduced in an effort to determine if the the variable-length particles suffered from *bloat*, which is a common problem in other Evolutionary Algorithms that use variable-length vector representations. No evidence of bloat was found.

Based on an overall comparative review of the both the fixed-length and variable-length forms of Grammatical Swarm it is recommended that the simpler fixed-length Grammatical Swarm with particle representation sizes of 200 codons in length be adopted.

Supervisor: Dr. Michael O'Neill

‘Would you tell me, please, which way I ought to go from here?’

‘That depends a good deal on where you want to get to,’ said the Cat.

‘I don’t much care where —’ said Alice.

‘Then it doesn’t matter which way you go,’ said the Cat.

— *Alice’s Adventures in Wonderland*, by Lewis Carroll (1865)

Acknowledgements

There are a number of people who deserve my special gratitude for their support and guidance during the past year.

I would like to express my deepest gratitude to my supervisor, **Michael**. You gave up much of your valuable time to offer your expertise and assistance in every way you could throughout the year.

To my girlfriend **Jane**, I can not thank you enough for your all your help. Words are just not enough to express my heartfelt gratitude for everything you have done for me over the past year.

To **Mark**, a special thanks for helping me in every possible way with that seemingly endless string of projects and exams. It is finally all over!

To my **Family** for their support over the past year and not least, to the *swarm* of **friends** back home in Co. Clare.

Contents

Abstract i

Acknowledgments iv

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Objectives . . . 3

1.2 Methodology . . . 4

1.3 Contributions . . . 4

1.4 Overview of the Thesis . . . 5

2 Scientific Background 9

2.1 Evolutionary Computation . . . 9

2.1.1 Evolutionary Algorithms . . . 10

2.1.2 Evolutionary Strategies . . . 15

2.2 Evolutionary Programming . . . 16

2.3 Genetic Programming . . . 16

2.3.1 Genetic Algorithms . . . 17

2.4 Grammatical Evolution . . . 18

2.4.1 Biological System Metaphorical Approach . . . 19

2.4.2 The Concept of a Grammar & Backus Naur Form . . 23
2.4.3 The GE Mapping Process 26
2.5 Particle Swarm Optimisation 32
2.5.1 Background 33
2.5.2 The Basic Particle Swarm Algorithm 34
2.5.3 Parameter Selection 37

3 Social Programming 39
3.1 Introducing Grammatical Swarm 39
3.2 GS Parameter Selection 40
3.2.1 Maximum Velocity, V_{Max} 40
3.2.2 Dimension and Velocity Capping, C_{Min} & C_{Max} . . . 41
3.2.3 Real Valued Vectors 41
3.2.4 Fixed Length Particles 42
3.2.5 The Wrapping Operator 43
3.2.6 Other Parameter Settings 43
3.3 GS - Program Generation 44
3.3.1 PSO - Search Mechanism/Engine 44
3.3.2 GE - Mapping Process 46
3.3.3 The Fitness Function 46
3.4 Grammatical Swarm Comparative and Verification Study . . 47
3.4.1 The Problem Domains 48
3.5 Experiment and Results 54
3.5.1 Discussion 57

4 Pseudo-Random Number Generator Investigations In Grammatical Swarm 59
4.1 Introduction 59
4.2 Pseudo-Random Number Generators 60
4.2.1 The C/C++ System-Supplied PRNG - `rand()` . . . 61

4.2.2 An Implementation of the Mersenne Twister PRNG -*eoRng* . . . 64
4.3 Experimental Settings . . . 67
4.3.1 Experiment: The Effects of Two different Quality PRNGs on GS . . . 67
4.4 Discussion . . . 73

5 Variable-Length Grammatical Swarm **75**
5.1 Introduction . . . 75
5.2 Particle Size Investigations in Fixed-Length GS . . . 76
5.2.1 Results . . . 78
5.3 Introducing the Variable-Length Grammatical Swarm Representation . . . 83
5.3.1 Initialisation . . . 83
5.3.2 Variable-Length GS Strategies . . . 84
5.4 Proof of Concept Experiments and Results . . . 86
5.4.1 Experiment A: A Variable-Length GS Initialised with 100 Codons . . . 86
5.5 Results . . . 87
5.5.1 Experiment B: A Variable-Length GS Initialised with 200 Codons . . . 91
5.6 Results . . . 93
5.6.1 Experiment C: A Comparative Analysis of the 100 and 200 Codon Variable-Length GS . . . 97
5.7 Variable-Length GS Particle Size Evolution . . . 99
5.7.1 Results . . . 101
5.8 Discussion . . . 105
5.8.1 Fixed-Length Particle Size Investigation . . . 105
5.8.2 Variable-Length GS 100 Codon Implementation . . . 106
5.8.3 Variable-Length GS 200 Codon Implementation . . . 107

5.8.4 Variable-Length GS Comparative Analysis 109
5.8.5 The Evolution of Size in the Variable GS 109

6 Conclusion **112**
6.1 Summary 112
6.2 Future Work 115

Bibliography **123**

List of Figures

2.1 A comparison between GE approach and the molecular biological process of *transcription* and *translation*. 20

2.2 Example Grammar for Generating Numbers 25

2.3 Simple Mathematical Expression BNF Grammar 27

2.4 This shows an individuals' *genome*, represented both in binary and in decimal format. 29

2.5 PSO Velocity Update 36

3.1 The Grammatical Swarm process, illustrating 1) the concept of a *swarm* of particles(fixed length vectors) in a *search space*. and 2) the mapping of the sample BNF grammar to produce a candidate solution 45

3.2 The toroidal grid for the Sante Fe Ant Trail problem. 49

3.3 Plot of Quartic Symbolic Regression, $f(a) = a + a^2 + a^3 + a^4$ on the interval [0,1]. 51

4.1 Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which showing the mean fitness(left) and the cumulative frequency of success(right) on tackling the Santa Fe Ant Trail problem. 69

4.2 Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of sauces(right) on tackling the Quartic Symbolic Regression problem. 70
4.3 Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of success(right) on tackling the 3 Multiplexer problem. 71
4.4 Plot of the fitness achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of success(right) on tackling the Mastermind problem. 72

5.1 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the `100` codon and `200` codon GS implementations on tackling the Santa Fe Ant Trail problem. . . . 79
5.2 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the `100` codon and `200` codon GS implementations on tackling the Quartic Symbolic Regression problem. 80
5.3 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the `100` codon and `200` codon GS implementations on tackling the 3 Multiplexer problem. 81
5.4 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the `100` codon and `200` codon GS implementations on tackling the Mastermind problem. 82
5.5 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Santa Fe Ant Trail problem. 87

5.6 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Quartic Symbolic Regression problem. 88
5.7 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the 3 Multiplexer problem. 89
5.8 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Mastermind problem. 90
5.9 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Santa Fe Ant Trail problem. 93
5.10 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Quartic Symbolic Regression problem. 94
5.11 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the 3 Multiplexer problem. 95
5.12 Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Mastermind problem. 96

List of Tables

2.1 Number of Production Rule Alternatives 28

3.1 Truth Table for the 3 Multiplexor problem 53

3.2 A comparison of the results obtained for Grammatical Swarm and Grammatical Evolution extracted from the proof of concept paper [38] and the GS results obtained in this study(GS Verify) over 100 runs across all the problems analysed. 55

4.1 The experimental settings for both GS implementations on the PRNG investigation. 68

4.2 A summary of the results for each of the problems tackled by both PRNGs, showing the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem. 73

5.1 The experimental settings for the Fixed-Length GS particle vector size investigation. 78

5.2 The experimental settings for the Variable-Length GS particle vector size investigation. 86

5.3 The experimental settings for the Variable-Length GS particle vector size investigation for the 200 codon implementation. . 92

5.4 A comparison of the results obtained by the 200 codon and 100 codon particle initialised Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four strategies. 98

5.5 A summary of the results for the Fixed-Length Particle Size Experiments, showing the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* on tackling each of the problem domains. 106

5.6 A summary of the results obtained by the [1, 100] particle initialised Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four Variable-Length GS strategies. 107

5.7 A summary of the results obtained by the $[1, 200]$ particle initialised (200 Codon) Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four Variable-Length GS strategies. 108

5.8 A comparison of the results obtained by the best Variable-Length GS and best Fixed-Length GS. It shows the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four strategies. 110

Chapter 1

Introduction

One model of social learning that has attracted interest in recent years is drawn from a swarm metaphor. Two popular variants of swarm models exist, those inspired by studies of social insects such as ant colonies, and those inspired by studies of the flocking behaviour of birds and fish. This study focuses on the latter. The essence of these systems is that they exhibit flexibility, robustness and self-organization [2]. Although the systems can exhibit remarkable coordination of activities between individuals, this coordination does not stem from a 'centre of control' or a 'directed' intelligence, rather it is self-organising and emergent. Social 'swarm' researchers have emphasized the role of social learning processes in these models [17, 16]. In essence, social behaviour helps individuals to adapt to their environment, as it ensures that they obtain access to more information than that captured by their own senses.

This thesis details a number of investigations into a recently developed automatic program construction algorithm based on a Particle Swarm learning model, called Grammatical Swarm (GS) [38]. In this GS methodology, each particle or real-valued vector, represents choices of program construction rules specified as production rules of a Backus-Naur Form grammar.

This approach is grounded in the linear Genetic Programming representation adopted in Grammatical Evolution (GE) [42], which uses grammars to guide the construction of syntactically correct programs, specified by variable-length genotypic binary or integer strings. The search heuristic adopted with GE is a variable-length Genetic Algorithm. A variable-length representation is adopted as the size of the program is not known a-priori and must itself be determined automatically. In the GS technique presented, a particle's real-valued vector is used in the same manner as the genotypic binary string in GE. This results in a new form of automatic programming based on social learning, which we dub *Social Programming*, or *Swarm Programming*. It is interesting to note that this approach is completely devoid of any crossover operator characteristic of Genetic Programming.

There are three primary investigations documented in this thesis; the first of these investigations attempts to replicate the results produced in the study by [38] with the implementation of a new GS algorithm i.e. it aims to reproduce a version the GS which is constructed so that it is identical in every aspect to the version presented in the GS proof of concept paper.

The second investigation performs a study into the effects of different quality Pseudo-Random Number Generators (PRNGs) on the GS.

The third investigation, introduces a novel variable-length Particle Swarm Algorithm for the automated construction of a program using a Social Programming model. The performance of this variable-length Particle Swarm approach is compared to its fixed-length counterpart on a number of benchmark problems.

1.1 Objectives

- To develop a GS algorithm identical in construction to that of the algorithm presented in the proof of concept paper. This replicated GS implementation will be constructed using an equivalent PRNG and identical values to those used in the original for each of the various algorithm settings. This aims to verify that both implementations are equivalent, thus ensuring that the investigations conducted in this study produce valid results.

- To develop two implementations of the GS using a different quality PRNGs for the production of random numbers where necessary in each, in order to determine if different quality PRNGs effect the performance of a GS algorithm.

- To investigate if an increase in the size of the fixed-length vector representations effects the performance of the algorithm, in this case doubling the fixed-length vector size.

- To develop a *novel* version of the Grammatical Swarm algorithm through modification of the structure primary components of the Particle Swarm learning algorithm by means of incorporating variable-length capabilities into the particle vector representations.

- To show how particle vector size evolves over the course of simulations in order to determine (1) if the vectors tend to *converge* at a certain size and (2) if particle *bloat* is an issue in the variable-length form of GS.

- To investigate if an increase in the size of the variable-length vector representations effects the performance of the Variable-Length GS algorithm i.e. the doubling of the *potential* size of a variable-length vector.

1.2 Methodology

This thesis builds on the work of existing algorithms. The original algorithms are presented and discussed firstly in terms of their background and composition and this is followed by a thorough description of the workings of the algorithms, accompanied with various implementation details and mathematical formula where necessary. This information is also reinforced by example and with the aid of diagrams.

The various investigations performed in this thesis were empirically analysed using four benchmark functions. The results obtained from the various experiments conducted are, in the majority of cases, presented in the form of graph plots and the results are generally summarised in tabular form to facilitate discussion.

Due to the stochastic nature of the population-based evolutionary experiments each experiment conducted were simulated exactly 100 times and the mean values over all the runs were calculated to ensure accuracy of the results.

1.3 Contributions

The following provides a list of the primary contributions of the thesis:

- The verification of the results in the original GS implementation presented in [38].
- The discovery that a poorer quality PRNG does in fact degrade the performance of the GS algorithm.
- The discovery that an increase in the size of the fixed-length vector

representations results in a significant improvement in performance for a number of different optimisation problems.

- Another contribution that this thesis offers is the development of a *novel* Variable-Length GS algorithm. The study demonstrates the feasibility of successfully generating computer programs using a variable-length form of the Grammatical Swarm algorithm on a diverse selection of optimisation problems. The study found that that the conventional bounded GS outperforms the Variable-Length GS, however it must be stressed that future investigations may find in the variable-length favour.

- An analysis of the behaviour of the interchangeable nature of the size of the vector representations throughout the course of the simulation on tacking the various problems.

- The discovery that an increase in the implementation size of the Variable-Length GS vector representations results in a significant improvement in performance for a number of different optimisation problems.

1.4 Overview of the Thesis

Chapter 2 provides a review of the *scientific background* of the thesis. Firstly it introduces the concept of optimisation. This is followed by an overview of the Evolutionary Computation methodology, presenting a dedicated explanation to each of the following *traditional* population-based search strategies, called Evolutionary Algorithms(EA): Evolutionary Strategies(ES), Genetic Algorithms(GA), Genetic Programming(GP) and Evolutionary Programming(EP). This is followed by a detailed description of another EA called Grammatical Evolution. GE is a novel EC method that can be used to produce computer programs or solutions in an arbitrary language. A description of the algorithms workings and its parallels with the

workings of biological systems found in living creatures in nature are highlighted. Finally, the Chapter presents a description of another relatively new optimisation method that is based on a social-psychological metaphor called Particle Swarm Optimisation. Note that an *emphasis* is given to both the GE and PSO algorithms as these form the basis of the work presented in Chapter 3.

Chapter 3, entitled *"Social Programming"* provides an overview of the *Grammatical Swarm (GS)* algorithm. GS is a form of Swarm/Social Programming and it is a hybrid algorithm consisting of a Particle Swarm learning algorithm coupled to GE which providing a method for the automatic generation of computer programs. A detailed description of the algorithm is provided by sectioning the algorithm into its various components and presenting an example program generation walkthrough. Next, four optimization problems that are considered benchmark standards in EC field are introduced. These problems were tackled by the various GS experiments presented in the remainder of the thesis. The final section of the Chapter documents the first of these experiments which had the objective of verifying the GS results presented of the original proof of concept paper.

Chapter 4 examines the performance effects of different quality Pseudo-Random Number Generators (PRNG) on the GS algorithm. An introduction to the principles of *randomness* focusing on the difficulty of producing numbers that are truly *random* using determinist method. Next, two different quality PRNGs are described, the first is the standard C/C++ `rand()` which is commonly described as being weak in the literature. The second presents an implementation of a leading PRNG in the scientific community called the Mersenne Twister. It has proven to be a *robust* and *stringent* method for producing random numbers and it passes all the various tests

for randomness. Furthermore, the results of a comparative experiment on the effects of two GS algorithms are presented, with each implemented using one of the aforementioned PRNGs. The Chapter concludes with a discussion of the experimental findings.

Chapter 5 details the researcher's novel implementation of a Variable-Length GS algorithm. Firstly, the results of an investigation conducted into the effects of modifying the canonical fixed-length GS are presented. This experiment involved increasing the vector-length of the population of *particle* representations to double the original size. This is followed by an overview of the Variable-Length GS, which outlines its implementation details and draws a particular emphasis to *four* implementation strategies which were necessary to incorporate variable-length particle dynamics in the new algorithm. The remainder of the Chapter documents the proof of concept experiments each presented in the form of comparative study between the various strategy implementations. They are listed as follows:

- An investigation into the performance effects of four different Variable-Length GS implementations.
- An investigation into the performance effects of using a Variable-Length GS implemented with particles that are initialised so that they can take on a potential vector size of double that of the size of the particles in the first Variable-Length GS implementation.
- A comparative analysis of the fixed-length and variable-length forms of GS is presented.
- An investigation is conducted into the evolution of particle size in each of the Variable-Length GS implementations, in an effort to obtain further information about the effects of the various implementation strategies.

The final section presents a discussion of the various experiments documented in the Chapter. In particular, it emphasises the performance of the novel Variable-Length GS incarnation in comparison to the canonical fixed-length (bounded) GS.

Chapter 6, presents a summary of the Thesis, focusing on the findings of the various investigations documented in Chapters 2-4. Finally, some topics that may be worthy of future research are discussed.

A **Bibliography** and an **Index** of the important terms, names, formula and various symbols used throughout the thesis is given at the end of this document.

Chapter 2

Scientific Background

This chapter discusses the scientific background of the thesis. It was compiled following an exhaustive search of the available literature. It aims to equip the reader with a sufficient grounding of the Evolutionary Computation (EC) methodology fundamentals. It gives a detailed description of the methodologies primary techniques. In particular, an emphasis is geared towards the Grammatical Evolution (GE) evolutionary algorithm and the Particle Swarm Optimisation (PSO) model. Overall, it serves as an introduction to Grammatical Swarm (GS) algorithm which is presented in the following Chapter entitled *Social Programming*. The GS algorithm constitutes a Particle Swarm algorithm coupled to a GE evolutionary algorithm, thus justifying the chapters emphasis on both of these algorithms.

2.1 Evolutionary Computation

In the late 1850's, Charles Darwin published a book entitled *The Origin of Species* [7] which introduced the modern theory of evolution through natural selection. His book was very influential resulting in the popularisation of his theory which has been accepted as the dominant scientific explanation of diversification in nature. Natural selection is the fundamental concept

underlying his theory of evolution . This is a biological process that affects the inheritance of individual traits from generation to generation. It results in the modification of an entire population over time such that individuals in successive generations gradually adapt to their environment.

About a century later, a number of methods were designed independently by various computer scientists, that simulated the principles of Darwinian evolution and natural selection inside a computer. These *Evolutionary Computation* simulations, more commonly referred to as Evolutionary Algorithms (EAs) have since grown in popularity. EAs are powerful, efficient and adaptive search mechanisms that have been used with much success for a variety of applications such as, machine learning, design, classification, automatic program generation, etc. They have proved to be particularly successful for solving combinatorial optimisation problems , in most cases, achieving better results than the more conventional methods.

The following section will present some of the core EAs that are in existence. Firstly, an introduction is provided in the form of an overview of the history of the core EAs. This is followed by a formal description of a generic EA. Finally, the section is concluded with a more detailed description of the core EAs.

2.1.1 Evolutionary Algorithms

History of Evolutionary Algorithms

The most successful of the Evolutionary Algorithms (EAs) include Evolutionary Strategies (ES), Evolutionary Programming (EP), Genetic Algorithms (GAs) and Genetic Programming (GP). One of the first EA was introduced in 1973 by Ingo Rechenberg [46]. He developed the ES and his algorithm was primarily used for numerical optimisation. Their simulations

focused on the propagation of behaviour traits using selection and mutation in an effort to find an optimal solution to a given problem. Shortly afterwards, Laurence Fogel , introduced EP which used a tree representation to solve finite state machines. EP emphasized the propagation of behavioral traits using the entire species in the evolutionary process, as opposed to the ES technique where evolution was based at an individual level. In the mid 1970's the Genetic Algorithm was introduced by John Holland [15] which was similar to ES but with one primary difference; it used a crossover operator to evolve individuals. Much later, in the year 1992 the concept of Genetic Programming was popularised by John Koza [18]. The fundamentals of GP were inspired by the EP technique and it was used to evolve parse trees of statements. Recently, GP was extended with the introduction of a grammatical approach to Genetic Programming, and this was called, Grammatical Evolution(GE). GE will be discussed in detail in Section 2.4.

Formal Algorithm Representation

Formally, an EA can be generally characterised by Algorithm 1 shown below. This algorithm was adapted from [1] and a full description of its workings is presented in the paragraph which follows.

Algorithm 1: General Evolutionary Algorithm

1: $t \leftarrow 0$
2: $P(t) \leftarrow$ initialise(μ)
3: $F(t) \leftarrow$ evaluate($P(t), \mu$)
4: **repeat**
   $P'(t) \leftarrow$recombine($P(t), \Theta_r$)
   $P''(t) \leftarrow$mutate($P''(t), \Theta_m$
   $F(t) \leftarrow$evaluate($P''(t), \gamma$)
   $P(t+1) \leftarrow$select($P''(t), F(t), \mu, \Theta_s$)
   $t \leftarrow$t+1
   until *Stopping Criterion*;

Given a population of μ individuals, $P(t) = (x_1(t), x_2(t), ..., x_\mu)$ at time t, where each individual represents a candidate solution to a given problem in a search space S i.e. $x_i \in S$. The fitness of each of these candidate solutions is determined by applying a fitness function, $f(x)$ to each individual in $P(t)$. Thus, the fitness of the entire population can be calculated from, $F(t) = (F(x_1(t)), F(x_2(t)), ..., F(x_n(t)))$. The following explains what each of the remaining *strategy* parameters represent:

μ , is the the total population size including parents and offspring.

γ , is the number of individuals in the current population.

Θ_r, Θ_m and Θ_s are the recombination, mutation and selection operators, respectively. Each of these specify the probability of applying the corresponding operator.

$P''(t)$, denotes the new population, reduced to the size of the parent population, μ.

This generic EA intends to show how the process of evolution is modeled inside a computer. An EA like this can be used to find a solution to an optimisation problem. The EA is initialised by scattering a population of individuals(μ) throughout a conceptual landscape. An individual is a structure that describes a potential solution to the given problem. This is achieved by encoding the problem variable into the genome of each individual in the population. Each individual also has a fitness value which is a measure of how close the individual is to the solution.

The population of individuals is evolved over a series of consecutive time steps, each of which is called a generation . At each of these generations all individuals in the population are subject to a fitness function, which serves to rank each individual according to its worth in the particular environment.

In other words, each individual is evaluated in terms of the solution that it describes such that, if a good quality solution is derived then that individual is assigned a fitness value that corresponds to the quality of that solution. Once the fitness(F) of the entire population is computed the population is evolved and this achieved by applying the following evolutionary operators to the population.

Selection: In the context of an EA, selection, identified in the generic algorithm by the selection operator,Θ_s is a combination of reproduction and selection. Its aim in this context is the same as in natural selection i.e. to adapt a population to its environment. The process removes the weaker individuals from the population thus, allowing the more successful ones to reproduce and consequently propagate their genetic material onto subsequent generations. The expression 'survival of the fittest' sums up the selection operators intent. Selection is responsible for determining which parents will breed, the number of offspring to produce and selecting the individuals that are deemed to be too weak to survive into the next generation. A significant number of selection methods have been proposed over the years, the most prominent being variants of *tournament selection* and *roulette selection*.

Recombination: Recombination is sometimes referred to as crossover. Crossover is the process of selecting two or more *parent* individuals and combining them together to produce a new individual offspring with contains some of the characteristics of both individuals. This introduces the concept of a generation where if the parents generation is denoted by N, the offspring's generation is then $N + 1$ and $N + 2$ will denote the next generation after that and so on for each subsequent generation. Typically, the parents are removed to allow the new offspring to propagate their genetic material. The rate at which crossover occurs is defined by the recombination operator probability, Θ_r. There are a

number of variations of crossover to include Single Point, Two Point, Uniform and Arithmetic. Each of these vary in the method in which they select sequences of information from the parents, to recombine to form an offspring.

Mutation: During the evolutionary process of an EA a populations individuals can sometimes converge on a solution too early and as a results the algorithm gets trapped at a *local optimum*, a situation which is referred to as *premature convergence*. One of the primary causes of premature convergence is the lack of diversity in the population i.e. the individuals in the population are too similar to each other.

A *consequence* of using crossover is that it reduces the population diversity. Recombination does ensure, that the fitter individuals genetic material propagates down through the generations but this means that in each of the subsequent generations the populations individuals gradually becoming more alike which results in the loss of the *unique* characteristics. This may be desirable in some ways as the unique characteristics may weaken the individuals, however these characteristics may be needed to improve or even solve the problem in later generations.

This problem is resolved with the introduction of the mutation operator, Θ_m. The mutation operator increases genetic diversity in the population, by randomly altering a small proportion(defined by Θ_m) of an individuals genetic material to produce a new individual [33]. This promotes *exploration* of the *landscape* thus helping to reduce the probability of premature convergence. Mutation is typically applied *after* crossover and for best performance the mutation should be set to a small value. Goldberg [10] maintains that best results are obtained by setting the mutation rate to the inverse of the total number

of chromosomes in the genes.

2.1.2 Evolutionary Strategies

ES were developed by Rechenberg [46]. The first ES which was developed was called the *(1+1)* or the *two-membered* ES. This used the principle of mutation to find a solution. In this implementation, normally distributed mutations are applied to each parent in the population, resulting in the generation of one offspring per parent. The two-membered ES was eventually extended with the introduction of the *multi-membered* or *(m+1)* strategy. In this strategy, the number of parents were increased to m which necessitated the use of a recombination operator.
There are two variants of this (m+1) strategy:
(μ, γ): This is called the *comma* strategy, where μ parents are used to generate γ offspring. In the population of the next generation, μ is selected only from the offspring, γ. This may result in good solutions being removed, since the parent population is destroyed but on the other hand, the diversity of the population is increased as there is a greater quantity of γ.

(μ+γ): This is called the *plus* strategy, where μ parents are used to generate γ offspring but in this case the μ parents and γ offspring are concatenated to form a new population. The best individuals from both μ and γ are then selected to be the parents for the next generations offspring.

There are two reproduction operators in ES, *Gaussian mutation* and *intermediate recombination*. The former, which is used in the (1+1) strategy, adds a random number generated from a Gaussian distribution to each element of an individuals vector (genome) to create a new individual. The latter, is used in the situation where two parents are used to produce one offspring i.e. (m+1). Elements from the vectors of both parents are added

together and an average is calculated, element by element which are used in turn to form an offspring.

2.2 Evolutionary Programming

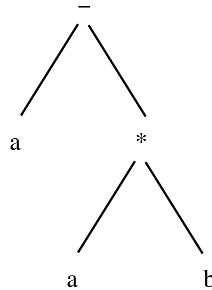
In the 1960's L.J. Fogel, introduced a new stochastic optimization strategy, called Evolutionary Programing (EP). The book "*Artificial Intelligence Through Simulated Evolution* [9]" which described the EP process received considerable attention at the time and it is regarded as a landmark publication in the EC field. This book described the process of evolving a finite state automata in order to predict a series of strings of symbols. EP is similar to ES but with one primary difference, EP does not exchange (genetic)material between individual representations i.e. it is completely devoid of any *recombination* operators. Instead, a *mutation* operators is used. A commonly used representation is that of a fixed-length real-valued vector. The process of selection in a *typical* EP is described as follows:

1. Populate a conceptual landscape with random solutions.
2. Select the individuals in the population to be the parents and mutate each to form N offspring.
3. Assess each member of the entire population based on its fitness. Use some form of stochastic tournament to determine a number of survivor solutions that will form the next generation.

2.3 Genetic Programming

Genetic Programming (GP) was conceived by J. Koza [18] in 1992 and it was inspired by EP. In GP, individual representation are not in the form of a fixed-length linear genome, instead an individual is represented as a variable-sized tree of values. These trees are used to construct computer

programs that once compiled and executed describe a potential solution to the given problem. For example, the program which describes a simple expression a-a*b is shown in the following tree structure:



GP is similar to a GA in that, typically the same variety of reproduction operators[1] are used in both algorithms to evolve the population, however in this case the operators are tailored so that they can be applied to the tree representations. For example, a common crossover operator used in GP is the *subtree crossover*. This allows for subtrees to be swapped between two parent tree representations. The parents are selected based on their evaluated fitness. The *closure* principle [20] ensures that on applying operators to the populations will not result in an illegal tree structures by permitting only structurally equivalent subtrees to be swapped.

2.3.1 Genetic Algorithms

The Genetic Algorithm (GA) was originally developed by Bremermann [3] and later popularised by J.H. Holland [15] in the 1970's. Hollands original experiments investigated the effects of natural adaptation in stochastic search algorithms which resulted in the development of the Schema [14]. Today, the GA is the most popular form of EC. The conventional GS uses a fixed-length string of binary values called *individuals* to represent candidate solutions. A population of these binary individuals are refined over generations through the use of various reproduction operators described in section

[1]Mutation is one exception to this as it is generally not used in GP.

2.1.1. An individuals vector structure is commonly described as being analogous to that of the genome, with each element having the same purpose as a gene of a biological organism.

Recombination occurs in the traditional binary string GA in the form of *bit-string* crossover. This process involves selecting two parents(based on evaluated fitness) , then selecting a crossover point and swapping sequences of binary information(chromosomes) to produce two new offspring. Mutation in the binary representation GA is typically implemented using the *bit-flipping* method. As its name suggest, it simply inverts a bit in an individual to form a new offspring. The parents are generally replaced by the new offspring.

2.4 Grammatical Evolution

For many years, it has been possible, through the use of EAs, to generate computer programs automatically (e.g.[18]). Although, there had been much success, researchers, in their experiments, would generally develop some self-tailored(bespoke) programming language to meet the needs of their particular problem. The fact that their experiments were bound to this specific language had obvious limitations. These limitations were, however, eventually overcome with the introduction of Grammatical Evolution (GE).

GE is a system that is capable of automatically generating and evolving computer programs in an arbitrary language. It is a relatively new addition to the evolutionary computation methodology, devised by researchers [48, 41, 37, 43] at the University of Limerick over six years ago. Since its inception, it has been successfully applied to problems in many different domains [5, 12, 34]most notably, it has enjoyed considerably success in Financial Modelling [39] applications.

It can be considered a form of grammar-based genetic programming.
GE is similar to GP in that both are capable of automatically generating compilable programs. GP, as described previously in Section 2.3, uses parse trees to represent a program. GE, however, does not employ the same technique, instead a Backus Naur Form (BNF) grammar is used to construct programs that are represented as a linear genome. Therefore, GE can be considered a form of grammar-based genetic programming.

The GE approach can be separated into two distinctive parts, 1) the search algorithm and 2) the mapping process. A Genetic Algorithm (see Section 2.3.1) is used as the as the primary search mechanism in the conventional GE. The mapping process consists of generating programs(solutions) by selecting rules(mapping) from the BNF grammar. The selection of rules is governed by the contents of the linear genome which is typically in the form of 1-dimensional binary vector. The GA maintains a population of these linear genomes, evolving each using the crossover and mutation reproduction operators. The contents of the linear genome and consequently the choice of BNF rules that are selected is determined by its position in the GA search space.

The following section describes the significance of the GE to the mapping process evident in molecular biology. This is followed by a description of the principles behind the BNF grammar notation. Finally a detailed overview the mapping process of GE is presented, using an example to show how GE can be used in the construction of a simple mathematical expression.

2.4.1 Biological System Metaphorical Approach

GE was derived largely from a biological metaphor and the workings of the approach can be communicated effectively using this analogy. GE employs

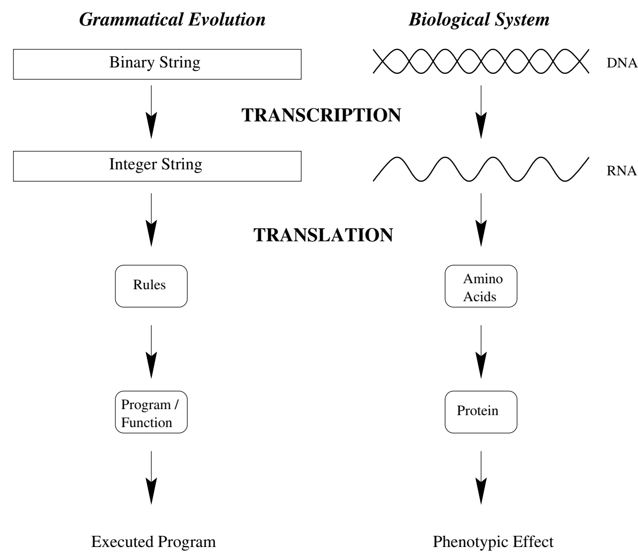


Figure 2.1: A comparison between GE approach and the molecular biological process of *transcription* and *translation*.

a genotype-phenotype mapping similar to that of the molecular biological processes of *transcription* and *translation*. The following is a comparison between the steps involved in generating a program using a GE approach and the biological steps involved in generating a phenotype in a genetic system, as is found in all living organisms in nature. The diagram presented in Figure 2.4 illustrates this.

It is evident that there is a direct analogy between the GE approach and the process of creating a phenotype that occurs in a Biological System(BS). The following paragraphs aims to reinforce the information presented in the diagram above by giving a comparative description between the GE approach and that of a BS. The diagram presented above in Figure 2.4 illustrates the the direct analogy between the GE approach and the process evident in a Biological System(BS). The following paragraph aims to reinforce the information presented in the diagram above by given a comparative description between the GE approach and that of a BS.

1. **Binary String/DNA:**

BS - In nature, all types of plants, animals and even bacteria are composed of cellular units. A cell is the basic unit of any living organism. All cellular forms of life need some mechanism to pass their individual characteristics onto their offspring. This is achieved by copying genetic material usually in the form of Deoxyribonucleic acid (DNA) into every new cell produced by that organism. DNA can be considered a blueprint for the individuality for a living organism. The function of DNA is to generate proteins; which are essential to the structure, function, growth and development of cells. Thus, the life, growth and individual traits of a living organism are all described by its DNA.

GE - The string of binary digits used to represent the genotype in GE are functionally equivalent to that of DNA in a biological context. DNA stores the genetic information that is used to determine the creation of proteins. The binary string used in GE has the same objective, as it also stores information that determines how a program is to be constructed.

2. **Integer String/RNA:**

BS - Consider a DNA to be similar in structure to a ladder where each rung of the ladder is called a nucleotide. There are four types of nucleotides, Adenine(A), Tyrosine(T) , Guanine(G) and Cytosine(C). A sequence of three rungs of the ladder(three nucleotides) is called a codon(e.g ATC). These codons first need to be transcribed into a slightly different format before it can be used to generate a protein. In a process called transcription, a cell synthesizes segments of DNA strands, called genes, into an RNA polymer. There are three types of RNA named, rRNA ,

tRNA and mRNA with the most significant being mRNA (messenger RNA). The purpose of mRNA is to deliver the information transcribed from the DNA to a cell organelle called a ribosome. A ribosome can be considered as a protein production factory.

GE - The computational equivalent of transcription is achieved in more simplistic manner than that of the biological process, described above. In the GE binary string there are only two nucleotides, namely, 1 and 0, unlike a BS where there are four such nucleotides that constitute a DNA molecule. The binary string is sectioned into groups of 8 nucleotides(bits) (e.g.`1010101`), which is also called a codon and each of these (binary) codons is then represented in integer format.

3. **Production Rules/Amino Acids:**

BS - In the ribosome, the sequence of codons in the mRNA, is used to specify the building blocks of proteins. These building blocks are known as amino-acids. The sequence of codons on the mRNA specifies the formation of amino acids in the protein that is to be encoded. The particular arrangement of amino-acids makes up a specific type of protein. Therefore, using this process, it possible to produce any type of a protein depending on the arrangement of amino acids.

GE - The codons(in integer format) contain the necessary information to select production rules from the BNF grammar. The sequence in which the production rules are selected determines the structure of the newly generated computer program. It is important to note that a different sequence of the same codons will produce a different program. This makes it possible to construct a variety of programs from the same grammar.

4. **Program/Protein & Executed Program/Phenotypic Effect**
   These stages are self-explanatory therefore they do not warrant a detailed explanation. They show the results of the both the GE and BS processes. In the case of GE, a fully compilable and syntactically correct program is generated and in the case of a BS, a full three-dimensional protein structure is created. The program is executed to produce some specific results, usually in the form of a solution to a particular problem. Similarly, the protein will be used to achieve some specific function within a phenotype.

The following subsections provides a more detailed description of GE, starting with an overview of the BNF notation. This is followed by a program generation example in the form of a step by step walkthrough, showing how a simple syntactically correct mathematical expression can be generated using a GE approach.

2.4.2 The Concept of a Grammar & Backus Naur Form

Avram Noam Chomsky, one of the most influential linguist of the 20th century, first introduced the concept of formal grammars. His work was concerned with the production of a system that had the capability to build properly formed phrases of a natural languages such as French and English. The principle behind his work is that every natural language is composed of basic units, called words. Chomsky recognized that words that constitute a natural language can be classified into grammatical categories. Such that almost all words in a language falls into a specific category e.g. articles, nouns, pronouns, verbs, etc. A sentence can be produced from a template of these grammatical category words where words can be substituted by other words from the same category without corrupting the validity of the sentence.

Chomsky used a grammar to specify the rules that governed the correct production of a sentence. By following simple rules a valid sentence could be generated. He categorised four types of grammars: Unrestricted Grammars, Context Sensitive Grammars, Regular Grammars and Context Free Grammars. The true benefit of these formal grammars is that not only can they be used to formalise both natural spoken languages but they can also be used to generate computer programming languages. In fact, grammars can be used to describe just about anything[x], such as maths expressions, neural networks, graphs, etc. GE is primarily concerned with Context Free Grammars, although there is no reason why other grammars should not work.

The Backus Naur Form (BNF)notation is used to express context-free grammars. It was originated by John Backus and shortly afterwards it was improved by Peter Naur. The improved version was popularised, following its use in defining the successful Algol 60 [55] programming language, see [35]. BNF breaks down the grammar of the language into derivation rules which are generally referred to as production rules. These production rules allow for the generation of syntactically correct instances of the language. The production rules ensure that it is not possible to produce invalid instances of the language. In the case of GE, this means that only valid programs can be generated.

A BNF production rule takes the following format:

```
<N>::= a
```

The `<N>`, represents a symbol and '`a`', represents a series of zero or more `non-terminal` and `terminal symbols`. The `non-terminals` are typically delimited by the '<>', angle brackets and these can be broken down recursively to `terminal` symbols. A `terminal` symbol can never appear on the

left hand side of the production rule and it can not be broken down further as it is the basic unit of the language e.g. a word in a sentence or, a key word or variable in a computer program. The notation allows for different alternatives to be specified making it possible to select different options for a particular `non-terminal` and this is represented by the vertical bar or pipe symbol, '|'.

More formally, a BNF grammar is represented by the 4-tuple $\mathcal{N}_\mathcal{T}, \mathcal{T}, \mathcal{P}, \mathcal{S}$.

where,

1. $\mathcal{N}_\mathcal{T}$, is a finite set of symbols, called `non-terminal` vocabulary.
2. $\mathcal{T}$, is a finite set of symbols, called `terminal` vocabulary.
3. $\mathcal{N}_\mathcal{T}$ and $\mathcal{T}$ do not have any elements in common i.e. $\mathcal{N}_\mathcal{T} \in \mathcal{T} = \{\}$
4. $\mathcal{S}$ is called the *start symbol*, which is a member of $\mathcal{N}_\mathcal{T}$ i.e. $\mathcal{S} \in \mathcal{N}_\mathcal{T}$
5. $\mathcal{P}$, is a set of procedures called *Productions* or *Production Rules* that are used in the process of mapping members of the set of `non-terminals`, $\mathcal{N}_\mathcal{T}$ to the members of the set of `terminals`, $\mathcal{T}$.

The BNF notation is best explained by example. The following shows a BNF grammar that can describe an integer number of any size.

```
(A)  <Number>       ::=  <SingleDigit>               (0)
                     |   <Number> <SingleDigit>      (1)
(B)  <SingleDigit>  ::=  0    (0)
                     |   1    (1)
                     |   2    (2)
                     |   3    (3)
                     |   4    (4)
                     |   5    (5)
                     |   6    (6)
                     |   7    (7)
                     |   8    (8)
                     |   9    (9)
```

Figure 2.2: Example Grammar for Generating Numbers

This grammar allows one to construct either an individual digit or concatenating a series of single digits which can be used to describe any whole number. The start symbol of the grammar is the non-terminal <Number> in production rule (A). This production rule can become one of two alternatives, identified in the diagram by the numbers to the left hand side of the respective alternative i.e. 0 and 1.

If we want to generate a single digit we take the first alternative, (0). This alternative is the non-terminal symbol <SingleDigit> which is specified in production rule (B). It can become any one of the ten alternative digits i.e. 0-9.

In order to generate a multi-digit number we follow a similar procedure, again we commence by selecting the start symbol, <Number> in production rule (A), however, in this case we take the second alternative, (1). This alternatives consists of two non-terminal symbols, <Number> and <SingleDigit>. The non-terminal, <Number> can be used to preform a recursive mapping of itself. <Number> can be chosen an infinite amount of times, thus, providing the capability to select a infinite amount of <SingleDigit> non-terminals all of which, can in turn, map to digits.

2.4.3 The GE Mapping Process

A comparative description of the biological analogy between the GE mapping process and that of the mapping of a genotype to phenotype employed by biological systems in nature, was presented in Section 2.4.1. This was

[1] The numbers, delimited by parenthesis to the left of each rule are used for explanatory purposes only. They do not constitute part of the actual BNF grammar itself. The same applies to the production rule letters.

followed by an introduction to formal grammars and more specifically the BNF notation. The information presented in this section aims to clarify the GE process further by providing an indept walkthrough of its mapping process. This walkthrough shows how it is possible to construct a simple mathematical expression using the GE approach.

$\mathcal{N}_\mathcal{T}$ = {<expr>, <unaryop>, <binop>, <operand>, <power>}
$\mathcal{T}$ = {√, +, -, ×, ÷, 2 , 3, 4.0, (,), x, y }
$\mathcal{S}$ = {<expr>}

and the production rules, $\mathcal{P}$ are written as follows:

```
(A)  <expr>     ::=  <expr><binop><expr>   (0)
                 |   <operand>             (1)
                 |   <unaryop> (<expr>)    (2)
                 |   <operand><power>      (3)
(B)  <unaryop>  ::=  √
(C)  <binop>    ::=  +    (0)
                 |   −    (1)
                 |   ×    (2)
                 |   ÷    (3)
(D)  <operand>  ::=  x    (0)
                 |   y    (1)
                 |   4    (2)
(E)  <power>    ::=  2    (0)
                 |   3    (1)
```

Figure 2.3: Simple Mathematical Expression BNF Grammar

The aim of the GE mapping process is to construct a program or solution by using the information contained in an individual's binary genome. To show how this is achieved, we will take a sample genome and show how it is used to select rules from a BNF grammar. This BNF grammar shown below represents the rules that govern the construction of a simple mathematical expression. Three sets are listed - $\mathcal{N}_\mathcal{T}$, $\mathcal{T}$ and $\mathcal{S}$, which represent the set of `non-terminal` symbols, the set of `terminal` symbols and the start symbol, respectively. The expression produced as a result of selecting the production rules, $\mathcal{P}$ will contain only elements from the set of terminals i.e. an

expression produced from this grammar is one that is comprised of elements in the set VT. As in all BNF grammars, we commence with the selection of the start symbol, $\mathcal{S}$, in the case of this grammar, the `<expression>` symbol. The grammar consists of five production rules, identified by the letters A to E to the left of each rule. During the mapping process, there are a number of alternatives that can be selected for each of these production rules, apart from production (B), for which there are no alternatives therefore the `<unaryop>` non-terminal will always derive the $\boxed{\surd}$ symbol each time that it is mapped.

Production Rule	Number *of* Alt's
(A)	4
(B)	4
(C)	3
(D)	1
(E)	2

Table 2.1: Number of Production Rule Alternatives

Table 2.1, shown above, summarises the number of alternatives available per production rule. If we take the first rule from the grammar, identified by the letter, A, which describes the <expression> non-terminal, there are exactly five possible alternatives available i.e. <expression> can become either <Expression> <operator><expression>, <variable>, <unaryop><expression> <operator>), <variable> or <power>. Similarly rule B, has four alternatives which means that <binop> can be transformed into one of those four rules and so on for the remainder of the productions rules, (C), (D) and (E).

As mentioned previously, GE uses the information contained in an individuals' genome in the mapping of non-terminal to terminal symbols. The diagram in Figure 2.4 shows an instance of an individuals' genome. Em-

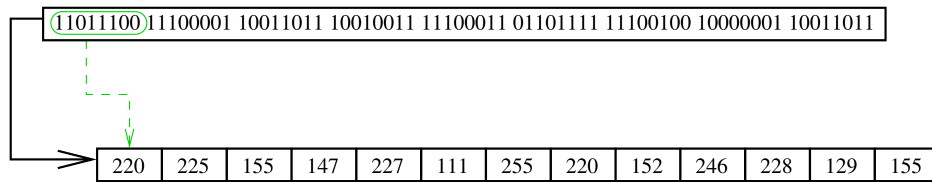


Figure 2.4: This shows an individuals' *genome*, represented both in binary and in decimal format.

phasized in this diagram is the division of the binary string(genome) into codons which is also shown beneath in decimal format. Of course, it is not necessary to convert the binary codons to integer format as binary values can be computed without effort by a computer; the integer representation is used only to facilitate ease of reading.

The mapping process in GE is concerned with determining which of the possible alternatives(if there are alternatives) in a production rule to select. The first codon in the diagram contains the binary digits - `11011100`, this converted decimal notation is the value 220. Taking the start symbol <`expression`>, or production rule `(A)`, we see that there are four possible alternatives (See Table 2.1) that can be derived. The following mapping function is used to determine which of these alternatives to select based on the genomes codon's value:

$$\boxed{Alternative = CodonValue \ \% \ Num.\ of\ \ Rule\ \ Alternatives}$$

where,

- Num. of Rule Alternatives: is the total number of rule alternatives for the current `non-terminal`.
- Codon Value: represents the value of the currently selected individuals' genome codon.

- %: represents the *modulus* operator.

Substituting the information obtained from the grammar gives,

$$\begin{aligned} Alternative &= CodonValue \ \% \ Num.\ of \ \ Rule \ Alternatives \\ &\Rightarrow 220 \ \% \ 4 \\ &\Rightarrow 0 \end{aligned}$$

This, gives us a value of 0, which means that the first alternative, `<expression> <binop><expression>`, is selected. This alternative contains three `non-terminals`, each of which must be mapped to `terminal symbols`. We continue the process taking the leftmost `non-terminal` from the chosen alternative which is another `<expression>` symbol. Then using the value of the second codon along with the total number of alternatives, we substitute them into the mapping function in order to determine an alternative for this current symbol. This give, $\boxed{225 \ \% \ 4 = 1}$, therefore we select alternative `(1)`, which means that we now have `<operand><binop><expression>`. Continuing the same process, we take the next codon value, $\boxed{155}$ to determine which one of the three operand to choose for the `<operand> non-terminal`. This gives, $\boxed{155 \ \% \ 3 = 2}$, therefore the `<operand> non-terminal` becomes alternative `(2)`, 4.0. This is a `terminal` symbol, so this cannot be mapped further, instead we move on to process the next leftmost symbol which is the `<binop> non-terminal`. Again, taking the next codon value, `147` and known that there are four possible alternatives for `<binop>`, we calculate $\boxed{147 \ \% \ 4 = 3}$. This gives us the $\boxed{\div}$ `terminal`. We follow the same procedure for the final `<expression> non-terminal`. $\boxed{227 \ \% \ 4 = 3}$, so this becomes `<operand><power>`. The next codon is $\boxed{110}$, giving $\boxed{111 \ \% \ 3 = 0}$, therefore we select the $\boxed{x}$ terminal. Finally, the `<power> non-terminal` becomes the value of $\boxed{3}$, calculated from $\boxed{255 \ \% \ 3 = 1}$. The expression generated as a result of applying the mapping process to the grammar is:

$$\boxed{4.0 \div x^3}$$

At times, during the translation process, it is possible to have a situation where genotypes that are not long enough to transform all the non-terminals into terminal symbols. If such an event occurs, a process called wrapping is used. Wrapping a genotype involves reusing the codons contained in the genotype. When the translation process has moved to the end(the right hand side) of the genotype (i.e. the last codon in the genotype is used) and there are still more codons needed to complete the translation the process is continued by simply moving to the first codon(the left-hand side) in the genotype, thus extending the quantity of codons until all non-terminals are transformed. An individual genotype can be wrapped a specified number of times.

The BNF grammar in the given example allows for the production of a number of simple mathematical expressions. However, this is done, purely to illustrate how a BNF grammar can be used, in the GE process, complete compilable programs are generated. The same technique as above is used but in that case the grammar is is tailored to produce code in a given language. Chapter 5, "*Variable Length Grammatical Swarm*" describes in detail some grammars that generate programs that were used in the experimental setup of this study.

The mapping process involves moving in a left to right direction along the genome[2], converting each of the binary codons into its corresponding integer values and then, using the mapping function, calculating the appropriate rule to selected from the BNF grammar. The process continues until one of the three situations listed below arises:

[2]However, this is not the case in a variation called πGrammatical Evolution [40]. This is a position-independent implementation of the Grammatical Evolutionary genotype-phenotype mapping process where the order of derivation sequence steps are no longer applied to `non-terminals` in the predefined fashion described above.

1. **Phenotype Produced**. All non-terminals have been mapped onto terminals. The result is that a fully compilable and functional program is generated. In the case of the given example, a complete and syntactically valid expression is generated. The program that is generated is called an *individual* or a *phenotype*.

2. **Wrapping**. This occurs when all codons in the genome have been used once and there are still non-terminal that need to be transformed to terminals. In other words, there are no more codons available to complete the transformations. The situation is resolved by invoking the wrapping operator. This allows for the remainder of the non-terminal to be transformed by returning to the beginning of the genome(left-hand side) and processing the same codons over again. The codons will continue to be processed until (a) either situation 1 occurs or (b) a the wrapping threshold is reached. The wrapping threshold is defined as the maximum number of wrapping events that can occur.

3. **Termination**. If the wrapping thresholds is reached and there are still rules that are not fully transformed, the mapping process is stopped. This typically results in the generation of an uncompilable program. To discourage the breeding of individuals with traits that result in termination the individual is assigned a minimal fitness value.

2.5 Particle Swarm Optimisation

There are many types of creatures found in nature that exhibit social behaviour in order to improve their situation or to solve problems. This social behaviour is particularly evident when creatures behave as a swarm, exchanging their best experiences in an effort to find an optimal solution to a problem. Popular every day examples are the swarming behaviour evident among social insects such as ants and bees or the flocking of birds and the

schooling of fish. Creatures that exploit the efforts of their neighbours i.e. those that behave as a swarm to solve problems are said to exhibit *swarm intelligence*.

2.5.1 Background

Since the early 1990's, researchers have been investigating into this swarming behavior, in an effort to model the underlying principles of a swarm and to use this to for function optimisation. Their efforts were rewarding and today there are models that have achieved much success. One such model that has been particularly successful is Particle Swarm Optimisation (PSO).

The PSO [17] model, is a relatively new variety of optimisation algorithms. It was first introduced in 1995 by a social psychologist, Kennedy and an engineer and computer scientist, Eberhart. It is inspired by the flocking behavior of birds and the schooling behavior of fish. Kennedy and Eberhart's model was strongly influenced by a swarming algorithm produced some years earlier by biologist Frank H. Heppner.

Heppner's [13] research into modeling the flocking behavior of animals, in particular birds. He developed an algorithm that consisted of a number of birds where each bird was programmed to search for a suitable roosting area. Upon initialization of the algorithm, the birds would form into flocks, and fly randomly throughout the solution space. If a bird in a flock flew above a good roosting area then it would either land or stay with the flock (depending on the weighting of certain parameters). Each bird tended to stay in the center of other birds in the flock. When one bird found a suitable roosting area the other nearby birds would adjust their trajectories and follow it. Thus, each bird would effectively pull the others to the solution.

Based on this, Kennedy and Eberhart realised that "social sharing of information among conspecies offers an evolutionary advantage" and as a result they developed the PSO model. However, instead of finding a suitable roosting area such as in Heppner's simulations the PSO model is capable of finding solutions to optimisation problems. Since its inception, PSO has had significant success in the research community. Most notably in the training of feed-forward neural networks. It has also proven to be comparable in performance with traditional evolutionary algorithms such as Genetic Algorithms (GA) [47].

2.5.2 The Basic Particle Swarm Algorithm

PSO is an adaptive algorithm that uses a population of individuals called particles for the optimization of continuous, non-linear problems. It is similar to evolutionary algorithms such as Genetic Algorithms (See 2.3.1) in the sense that it uses a population of individuals. However, unlike a GA, where the individuals in population are updated using principles of natural selection, the PSA maintains the same population and updates every individuals position at each iteration in an effort to find the best solution.

The Particle Swarm Algorithm (PSA) is initiated by populating an n-dimensional environment with particles. The solution to the problem is at an unknown location in this n-dimensional environment that is more commonly referred to as a problem space or search space. Each particle in the population describes a possible solution to the problem and this is determined by a particles location.

The particles are scattered throughout the search space during the initialization process i.e. they are randomly assigned locations. As well as a current location, each particle in the swarm has a velocity associated with it. The

value of a particle at a particular location in the search space is known as its fitness. As a particle moves closer to the solution its fitness increases. The fitness of each particle in the swarm is evaluated at the end of every iteration. This facilitates the comparison of different solutions to the given problem, each particles location being a potential solution.

Each of the particles maintain a memory of the best position in the search space that it has found to date, p_{best}. A global best position, g_{best} is also stored. This is the best position obtained so far by any particles in the swarm. In some variations of the algorithm, a neighbourhood topology is defined to constrain the interaction of particles. At every iteration each of the particles adjusts its velocity, calculating a new point where particles are moved, to examine. The velocity is influenced by the particles velocity at the previous time step(t-1), the location of pbest and gbest positions and some random parameters that will be described subsequently. Thus, at every time step, a particle is moved to a new position in the search space by computing a new point based on the particle's own history and the influence of the other members of the swarm.

The following provides a list of the steps involved in the algorithm.

Step 1. Initialize each particle in the population by randomly selecting values for its location and velocity vectors.

Step 2. Calculate the fitness value of each particle. If the current fitness value for a particle is greater than the best fitness value found by a particle so far, then revise p_{best}.

Step 3. Determine the location of the particle with the highest fitness and revise g_{best} if necessary.

Step 4. For each particle, calculate its velocity according to equation (1.1).

Step 5. Update the location of each particle.

Step 6. Repeat steps 2 - 5 until stopping criteria are met.

PSO Velocity Update Equation

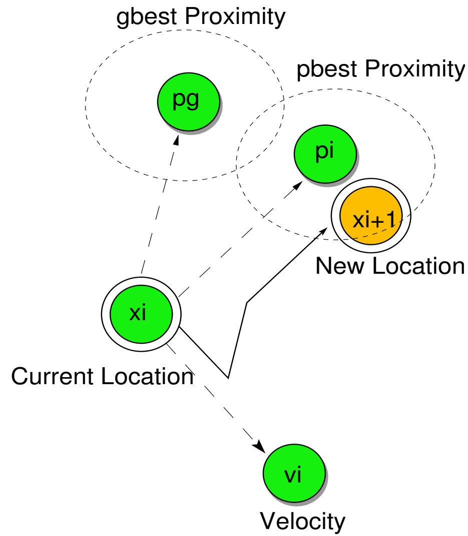


Figure 2.5: PSO Velocity Update

The velocity update , $v_i(t+1)$ is computed using the equation below:

$$v_i(t+1) = (\omega * v_i(t)) + (c1 * R_1 * (p_{best} - x_i)) + (c2 * R_2 * (g_{best} - x_i)) \quad (2.1)$$

This equation lies at the core of the PSO algorithm as it is responsible for the movement of the particles throughout the search space. Figure 2.5 above illustrates the updated process.
The other parameters are explained in the following sub-sections.

PSO Position Update

Once the velocity update for particle is determined, its position and p_{best} is updated if necessary.

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2.2)$$

$$y_i(t+1) = y_i(t) if, f(x_i(t)) \leq (y_i(t)) \tag{2.3}$$

$$y_i(t+1) = y_i(t) if, f(x_i(t)) > (y_i(t)) \tag{2.4}$$

Once the particles are updated it is necessary to determine if g_{best} needs to be updated. The following equation shows how this is achieved.

$$\widehat{y} \in y_0, y_1, \ldots, y_n | f(\widehat{y}) = max(f(y_0), f(y_1, \ldots, f(y_n)) \tag{2.5}$$

2.5.3 Parameter Selection

The Inertia Weight, ω

The inertia weight, ω:
where,

$$\omega = \omega max - ((\omega max - \omega min)/itermax) * iter \tag{2.6}$$

$wmax = 0.9, wmin = 0.4$ and $itermax$ is the total number of iterations and iter is the current iteration.

The inertia weight, commonly denoted by the ω symbol, was proposed by Eberhart and Shi [50, 49] to enforce convergence. It achieves this by controlling the momentum of the particles in the swarm. During the velocity update(calculation of v_i), the inertia weight ω is multiplied by the velocity at the current time step, v_i. Therefore, the value of ω determines the influence of the previous iterations velocity when computing the new velocity.

The control parameters, $c1$ and $c2$

It is necessary to balance both the social and individualistic factors associated with each of the particles in the swarm. We want to control the

extent to which the particles [move towards the direction of]learn from its own previous best(individuality) and the extent to which it learns from its peers(sociality) in the swarm. This is achieved through the use of two parameters, $c1$ and $c2$ control the weight of the personal best dimension and the global best dimension value, respectively. The c1 parameter, is the self confidence factor. The $c2$ parameter is the swarm confidence factor. This is shown in the velocity update equation (1). To introduce some variety or some chaos into the equation these weights are randomly altered using two more parameters, $R1$ and $R2$. These are random number calculated in the range $[0, 1]$.

Chapter 3

Social Programming

3.1 Introducing Grammatical Swarm

The term Grammatical Swarm (GS) refers to a novel Evolutionary Algorithm that is based on the principles of social learning. GS, introduced in 2004 by O'Neill and Brabazon [38], can be considered a form of *Social Programming* or *Swarm Programming* as it is based on an algorithm inspired by the swarming behaviour evident in nature (Section 2.5). The GS algorithm is a *hybrid* algorithm consisting of a Particle Swarm learning algorithm coupled to a Grammatical Evolution (GE) genotype-phenotype mapping to generate programs or solutions in an arbitrary language.

GE (Section 2.4) can be separated into two components 1) the search mechanism and 2) program generation. The conventional GE uses a Genetic Algorithm (GA) as its search mechanism. In this conventional implementation a population of individuals are scattered throughout a conceptual landscape, evolving to produce fitter individuals based on the natural selection metaphor using reproduction operators such as crossover and mutation. Individuals are represented in the conceptual landscape in the form of *binary* linear genomes. The information contained in these linear genomes is then

used to govern the selection of rules from a BNF grammar which in turn produce a functional solution usually in the form of a compilable program. GS, is similar to the implementation of GE in that it also constitutes a search mechanism and a program construction mechanics, but GS has one primary alteration in that it does not use a Genetic Algorithm to fulfill its search capabilities. Instead, GS adopts a Particle Swarm Optimisation learning strategy e.g. the GS algorithm is completely devoid of any reproduction operators.

This chapter describes in detail the working of this GS algorithm by presenting an in-dept description of its implementation. Firstly, an overview of the various parameters and properties of the GS search engine are presented. This is followed by a detailed walkthrough of the GS approach demonstrated with the use of a simple sample optimisation problem. This is followed by the a review of the comparative investigations conducted in the GS proof of concept paper [38]. These investigations use a number of combinatorial problems which are also described in detail in this section. The comparative investigation also documents the results of the first experiment of this thesis. This experiment indents to verify the integrity of the both the original GS and the GS that is used in the various experiments documented in the remainder of the thesis.

3.2 GS Parameter Selection

3.2.1 Maximum Velocity, V_{Max}

V_{Max} is applied to the each *particle* in the swarm at every iteration in order to control the maximum distance a particle can move in a single time step. In the GS implementation the maximum velocities V_{Max} are bound to value of $\pm$ 255. This means that a particle can move a *maximum* of 255 in either

a positive of negative direction.

3.2.2 Dimension and Velocity Capping, C_{Min} & C_{Max}

Represented in a programming language, a particle is typically made up of a number of vectors(see Code) with each of these vectors consisting of a number of elements. In the GS algorithm, there is a restriction placed on each of the vector elements so that a particle can only contain values within the range [0, 255]. The lower bound, 0 and upper bound, 255 are referred to as C_{Min} and C_{Max}, respectively. This is implemented after the new velocity has been calculated and added to the particles current vector, x_i to the compute the particles next position, $x_i(t+1)$ in the search space. If $xi(t+1)$ has a value greater than the value defined by C_{Max} then that value simply becomes C_{Max}. Similarly, if the value is smaller than C_{Max} then $x_i(t+1)$ is increased to C_{Min}. The following shows how this is implemented in C++ code(lines 4 and 5).

```
1:      for(int i=0;i<n_particles;i++) {
2:          for(int d=0;d<n_dim;d++) {
3:            p[i].next[d]=p[i].current[d] + p[i].velocity[d];
4:            if(p[i].next[d]>cmax) p[i].next[d]=cmax;
5:            if(p[i].next[d]<cmin) p[i].next[d]=cmin;
6:          }
7:       }
```

3.2.3 Real Valued Vectors

In the conventional GE or more specifically the GA search engine employed by GE, a population of individuals in the form of variable length binary strings are used to search for the optimal solution. GS does not follow a similar approach, instead each particle(equivalent to a GA individual) uses a vector of real-valued numbers in the range [0, 255]. However, the GE

mapping function can only accept *whole numbers* but the velocity update function contains parameters that hold floating point numbers; for example $R1$ and $R2$ are random numbers in range [0,1]. This can be seen in the following PSO equation, which is described in full in the previous Chapter.

$$v_i(t+1) = (\omega * v_i(t)) + (c1 * R_1 * (p_{best} - x_i)) + (c2 * R_2 * (g_{best} - x_i)) \quad (3.1)$$

As floating point multiplied by a whole number a floating point number will always be produced in velocity calculations. Therefore it is necessary to round the floating point values up or down to the nearest integer value. Thus, ensuring that only whole can be used in the mapping process.

3.2.4 Fixed Length Particles

Each particle in the population contains a number of vectors which describe its current and next location and velocity. In the PSO the number of dimensions is typically set according to the particular problem it intends to optimize. For example, consider the following equation $\boxed{2x + 1y = 4}$, then each particles' vectors would contain two elements, one to hold a value for variable x and one for variable y. However, when a particle describes an interchangeable number of rules that may be selected during the mapping process, it makes it difficult to set the total number of dimensions. In the GS implementation, all these vectors are of a fixed length and in the case of this implementation vector is constrained to 100 dimensions. The conventional GE uses variable length individuals in its GA search engine. Each individuals' binary string can increase or decrease in size which is adjusted at each generation. Thus, the genome length of the fitter individuals propagates to subsequent generations.

3.2.5 The Wrapping Operator

In the GE mapping process it is possible to have a situation where there are not enough codons to map all the non-terminal symbols to terminals. This will generally result in the generation of an invalid individual or a program that will not compile. Chapter 2 already described how this problem is overcome by wrapping the genome. In GS, this wrapping operator is also used; allowing a vector's elements to be used more than once. A situation may also occur where all non-terminals are mapped to produce a full program before the end of the genome is reached, thus leaving a surplus of unused codons. These extra codons(considered introns) are ignored and although they are of no use in this particular mapping the may be used in subsequent iterations.

3.2.6 Other Parameter Settings

For each of the experiments conducted in this thesis the following settings are adopted and any deviations will be noted.

Population Size A swarm of thirty particles is used to populate the search space.

Social Learning Factors C1 and C2 are set to the value of 1.

The Stopping Criterion The particles will continue to search for a solution for 1000 iterations. If a solution is found before the last iteration is reached, the search will still carry on until the 1000 iteration if finished. This is not strictly necessary but it is done in an effort to obtain accurate readings when averaging over a number of runs.

Inertia weight The inertia weight, ω:

where,

$$\omega = \omega max - ((\omega max - \omega min)/itermax) * iter \tag{3.2}$$

$\omega max = 0.9, wmin = 0.4$, $itermax = 1000$ which is the total number of iterations and *iter* is the current iteration.

3.3 GS - Program Generation

This section intends to further clarify the workings of GS. A sample problem specification is given and a brief walkthrough will guide the reader through the phases involved in the construction and evaluation of a candidate solution program to solve the problem. It is anticipated that this will reinforce the readers understanding of the GS algorithm, in particular how it is used to solve problems.

This walkthrough will take a sample particle and use it to construct a solution to a simple problem. The problem is of a simple anagram which consists finding a particular word given a series of letters that once arranged in a certain order constitute that word. There are six letters are *a,e,i,c,t,s*. The target word chosen is *'cat'* i.e. the problem is considered as *solved* if a particle contains the correct information to produce a program from the BNF grammar to describes the word '*cat*'. This example is intentionally simplistic as it aims only to communicate the workings of GS.

3.3.1 PSO - Search Mechanism/Engine

Figure 3.1 illustrates the entire GS process of producing a sample solution. The cube in the upper left of the diagram illustrates the concept of a swarm of particles. More specifically, this cube represents a conceptual search space *environment*; shown within are vectors that represent a *swarm* of particles. PSO was also previously described in detail in Chapter 2 (2.5) therefore it is unnecessary to delve into its specifics again. Instead this example is con-

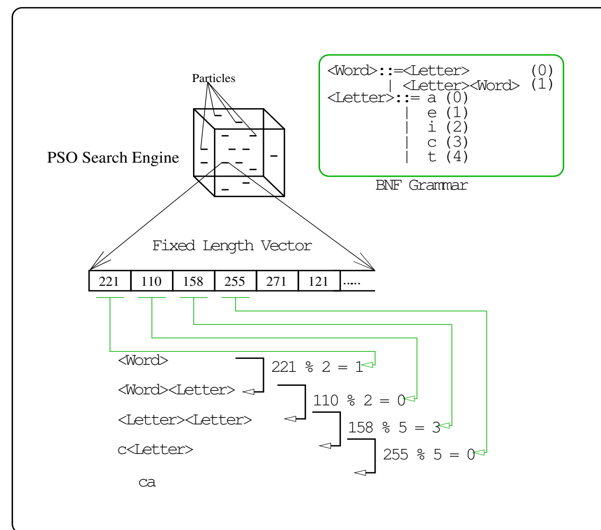


Figure 3.1: The Grammatical Swarm process, illustrating 1) the concept of a *swarm* of particles(fixed length vectors) in a *search space*. and 2) the mapping of the sample BNF grammar to produce a candidate solution

cerned about how the particles can be used to solve the given problem.

In the *canonical* PSO the values contained each of the elements of a particle's vector are typically inserted into the parameter space a defined optimisation problem (function). Consider the following example problem(function) e.g. $\boxed{2x^4 + 1y - c = 2}$. A PSO particle describing a potential solution to this problem would contain three elements i.e. element $0(x_0)$, element $1(x_1)$and element 2(x_2) would hold candidate values for the x, y and c parameters, respectively.

To evaluate the particle the values in the corresponding element of the particles' vector would be substituted into the parameter space and it would be evaluated by judging how close the result are to the target answer i.e. $\boxed{2}$. In the PSO search mechanism used by the GS, the values contained in the vectors are *not* used in the same way. The GS process is more complicated,

as these values do *not* slot directly into the parameter space of the function, instead they are used to govern the rule selection of a BNF grammar in the mapping process. Refer to the diagram which an instance of particle is used to illustrates this for the given problem.

3.3.2 GE - Mapping Process

The GE mapping process was described in Chapter 2, therefore it does not warrant another detailed explanation. Note that the GE process can be adopted directly without requiring any alteration, unlike the adoption of the PSO search engine, where it is necessary to adjust some of its parameters and even introduce new parameters (C_{Min}, C_{Max} (See Section 3.2.2 above)).

Referring to the sample problem, Figure 3.1 shows the BNF grammar for the problem in the top right and it shows in detail how each of the rules are mapped to produce a word using a combination of the terminal symbols. What is most important here is to understand how the PSO *plugs into* the GE mapping process by using a fixed length particle in the place of the *binary individual* of a GA, used in the conventional GE.

3.3.3 The Fitness Function

As the given problem is very simplistic, the fitness function can be effortlessly constructed. The purpose of the fitness function is to evaluate the particle in the swarm according to the quality of solution described. This is achieved by comparing the number and order of letters contained in the *candidate solution* to the target solution (i.e. '*cat*'). One point is awarded for each correct letter in the candidate solution string, regardless of order and an extra point is given if all three are in the correct order. If the string '*cat*' is matched then the *particle* that describes the solution is assigned 4 points or 100% fitness. In the GS process presented in Figure 3.1 the given particle

produced the word '*ca*'. Thus, the particle(shown in expanded form in 3.1) will be awarded a fitness of 0.5 or 50% of the solution is solved.

3.4 Grammatical Swarm Comparative and Verification Study

In the proof of concept paper the developers of GS demonstrated the feasibility of the automatic generation of programs using a PSO/GE hybridization approach. In *evaluating* their new algorithm they devised a series of experiments to rate the performance of the GS. In an effort to obtain a good approximation as to how well the GS is at automatically generating computer programs the experiments performed were in the form of a *comparative* study between the GS and that of the traditional Genetic Algorithm driven GE program generation approach. This thesis conducts a series of performance and modification investigations into the GS algorithm, each of which will be presented and analyzed in Chapter 4, *Pseudo-Random Number Generators* and Chapter 5, *Variable-Length Grammatical Swarm.* In order to conduct the various experiments, it was necessary to develop an implementation of a GS algorithm. The algorithm was developed using the various implementation details documented in the GS proof of concept paper. The *new* GS implementation was then used to tackle the various problem domains i.e. the same problems tackled by the original GS [38] and the performance results were recorded and then compared to the results of the original GS.

In summary, this section will provide the following:

1. **Introduce the problem domains**: Four different benchmark problems were tackled in these comparative experiments. These benchmark problems are well established as complex and sophisticated analysis techniques in the field of EC. Their purpose was to accurately measure

the GS algorithms performance potential. These problems warrant a detailed explanation as they are used extensively in the experiments documented in this thesis. There are four problems in total and they are detailed in the following subsection 3.4.1.

2. **Review of GS Performance**: A synopsis of the results and conclusions obtained for each of the experiments documented in the GS proof of concept paper is provided.

3. **GS Verification Experiment**: The results of a comparative study between the GS described in [38] and a replication of this GS, developed by the researcher are documented.

3.4.1 The Problem Domains

Problem 1: Santa Fe Ant Trail

The original ant problem was first introduced in the early 1990's for the field of Artificial Life and shortly afterwards it was adopted by Koza where it was used for GP performance testing [19]. It is now considered a standard benchmark in the GP field and it has proved to be a particular deceptive planning problem [24].

The ant problem consists of a 32×32 grid and an artificial ant which has the objective of finding pieces of food scattered randomly along a specific trail on the grid. The most common trail is the Santa Fe trail which consists of 144 squares with 21 turns. The gird is toroidal which means that it does not contain any boundaries such that, both the top and bottom and left and right sides are all connected. Once the ant enters a cell with a one value it is said to eat it. The ant has to collect(eat) 89 pieces of food in total in a number of time steps. The diagram in Figure 3.2 illustrates the toroidal grid; the highlighted squares represent the actual trail where a 1 value represents an instances of a food substance and a 0 represents the gaps on the

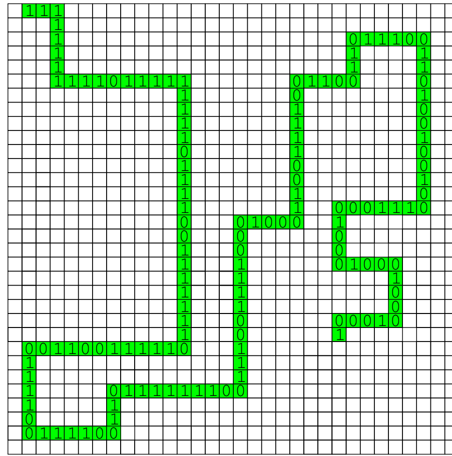


Figure 3.2: The toroidal grid for the Sante Fe Ant Trail problem.

trail that are empty i.e. there are 89 squares in total containing the value `1`.

The artificial ant starts at the top-left hand corner of the grid faced in the direction of the first food instance on the trail. The ant is represented in the form of a program solution. It has limited capabilities as it can only turn(90 degrees) left,right and move one square forward. It also has a seeking or lookahead function which allows it to look into the square it is facing for food. This lookahead function does not imposes an execution penalty whereas using each of the other functions costs one time step to execute.

To find a solution to the problem it is necessary to construct a program that contains the instructions to maneuver around the twisted trail to eat all 89 pieces of food. The fitness of a solution is determined by the calculating the total number of food substances that are consumed. The grammar used in this study is shown as follows:

```
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::= if(food_ahead){ <line> } else { <line> }
```

```
<op> ::=  left(); | right(); | move();
```

To clarify how the fitness of a solution is determined take the following example program.

```
if(food_ahead){if(food_ahead){move(); }else{right();} }else{right();}
```

This example represents a solution(ant) that once executed is capable of collecting one piece of food. Therefore, in the context of GS, a genome containing the instruction to select the production rules to construct such a solution would be awarded a fitness of 0.012 ($1 \div 89$) i.e. 1.2% of the entire problem is solved.

Problem 2: Quartic Symbolic Regression

The objective of a *symbolic regression* problem is to find a function that matches some unknown function on a specified interval. There are many types of symbolic regression problems that are commonly used in GA and GP fields, such as the Quartic polynomial, the Binomial-3 polynomial and the Rastrigin function. An instance of the *Quartic* polynomial [19] is tackled in each of the experiments conducted in this thesis.

The unknown function in a symbolic regression problem is commonly referred to as the *target* function, f(a)=y where a is the given input and y is the given output, called the independent and dependent variables respectively. In the case of the Quartic Symbolic Regression problem the target function is defined in equation 3.3 and a graph of the problem is also provided.

$$f(a) = a + a^2 + a^3 + a^4, a \in [0, 1]^{1} \tag{3.3}$$

[1] Range may vary in other studies.

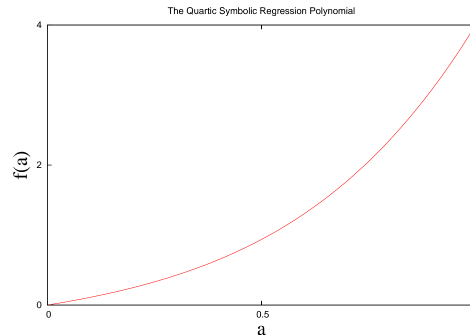


Figure 3.3: Plot of Quartic Symbolic Regression, $f(a) = a + a^2 + a^3 + a^4$ on the interval [0,1].

The problem is initialised by randomly generating one hundred numbers and inserting each in turn, into the parameter space (i.e. a) of the target function. These 100 random numbers are stored memory in a list which can be referred to as the input vector. The target function is evaluated for each of these random values to produce a list of 100 corresponding output solutions(y). This list is referred to as the input-output vector. The input-output vectors can be collectively viewed as a list (a_i, y_i), where a_i is drawn from the range $[0, 1]$.

The next step is to evaluate the candidate solutions. These input-output vectors are made available to each newly generated candidate solution. The fitness of a candidate solution is calculated by determining how similar the candidate function is to the target function. This is achieved by calling the candidate function 100 times, passing each of the values form the input vector, evaluating it to produce another (evolved) output vector.

The strength of a candidate solution (fitness) is determined by calculating the reciprocal of the sum of the absolute error of the evolved and target

output vectors.

The BNF grammar for this problem is shown below. A candidate solution (expression) can consist of a particular arrangement of operators (+,-,*,/) and an operand (a) symbol. Note, that there is no symbol in the grammar to express a number as a power, instead this calculation is provided by multiplying the operand the required number of times e.g. $2^2 = 2 \times 2$.

```
<expr> ::= <expr> <op> <expr> | <var>
<op>   ::= + | - | * | /
<var>  ::= a
```

Problem 3: 3 Multiplexer

A *multiplexer* is a logical circuit that takes information from a number of different inputs and combining them together to produce a single output. A 3 multiplexer is an instance of one, so called as it that takes three such inputs and routes them to a single output. A multiplexer can produce a different output depending of the boolean gates operated on the inputs. There are three types of boolean gates used in this problem 1) AND 2) OR and 3) NOT. An AND gates performs a logical and operation e.g. for three boolean inputs 1 0 1 each will be combined to produce an output of 1. The OR gate will produce an output of 1 if either of the three inputs are 1 otherwise it will produce an output of 0. A NOT gate simply inverts a bit.

The 3 Multiplexer problem is based in principle on the operation of these logical circuits. The objective of the problem is to discover an expression in *boolean* form that functions as a 3 multiplexer circuit given a list or Truth Table of inputs(3) and corresponding outputs(1). The inputs and outputs(target) for this problem are shown in the following Truth Table 3.1.

Test	I_1	I_2	I_3	O_1
(A)	0	0	0	0
(B)	0	1	0	1
(C)	0	0	1	0
(D)	0	1	1	1
(E)	1	0	0	0
(F)	1	1	0	0
(G)	1	0	1	1
(H)	1	1	1	1

Table 3.1: Truth Table for the 3 Multiplexor problem

There are eight test cases, identified in the table by the letters (A)-(H). The BNF grammar below is used to generate candidate multiplexers so that a correct arrangement of gates will produce the target outputs. The fitness is calculated by given a value of 1 to every evolved expression that returns the correct output using the three input cases. For example if the evolved boolean expression returns the correct output for four out of the eight gates then that candidate solution is assign a fitness of 0.5 ($4 \div 8$) i.e. exactly half the problem is solved.

```
<mult>  ::= guess = <bexpr> ;
<bexpr> ::= ( <bexpr> <bilop> <bexpr> ) | <ulop> ( <bexpr> ) | <input>
<bilop> ::= and | or
<ulop>  ::= not
<input> ::= inputs[0] | inputs[1] | inputs[2]
```

Problem 4: Mastermind

Mastermind is a code breaking problem consisting of a number of colored pins. The aim of the problem is for a code breaker to determine the correct combination of colored pins in a given solution. Each pin can be one of four different colors identified by the values 0,1,2 or 3 and there are eight pins

in the solution. This allows for a large number of different combinations which is why the problem is quite complex. In the case of the experiments evaluated in this study the given solution is | 3 2 1 3 1 3 2 0 |.

The fitness is calculated by awarding one point for each pin that has the correct color regardless of position; the code breaker earns an additional point to solve by guessing the correct order. Consider, the following example, where a candidate solution of 32132330 is produced. This will attain a fitness of 0.67 $(6 \div (8 + 1))$ as there 6 pins out of a total of 8 with the correct color but they are not in the correct order so a 6 out of a possible 9 points are scored for this particular solution. The grammar adopted to describe the solution can be written in one production rule; it is shown as follows:

```
<pin> ::= <pin> <pin> | 0 | 1 | 2 | 3
```

3.5 Experiment and Results

The results of this comparative study are shown in the following Table (4.2) which lists the results obtained from three algorithms on tackling the various problem described previously. The names of each of these problems are listed in the leftmost column of the table along with the results obtained for the corresponding algorithm. As shown in the table, three algorithms are compared. The first *two* algorithms listed, under each of the problem headings are the GS and GE implementation. The results from two these algorithms are extracted from the investigation conducted in the original proof of concept paper. The *third* algorithm listed, GS Verify, is the replicate GS. This algorithm was developed using the details provided in the original GS paper [38]. This study has two primary objectives, (1) to compare the results of the GS implementation that is used in the experiments in this study to that of the original GS implementaion (2) to ensure that

any of the experimental findings presented in the following Chapters are, in fact, legitimate. The highest fitness obtained by any particle over an average of one-hundred runs is listed under the heading, Mean Best Fitness where a rating of `1.0` would denote a successful solution and a rating of *0* is the lowest possible fitness that can be obtained. The mean average fitness results are listed in the second column and the total number of times that a *successful* solution was obtained over the one-hundred runs are listed in the last column.

	Mean Best Fit. (Std.Dev.)	**Mean Avg. Fit. (Std.Dev.)**	**Successful Runs**
S.F.A.T			
GS	0.85 (0.19)	0.38 (0.04)	43
GE	**0.90** (0.15)	**0.52** (0.13)	**58**
GS Verify	0.83 (0.19)	0.04 (0.01)	38
Q.S.R			
GS	0.31 (0.35)	0.07 (0.02)	20
GE	**0.88** (0.30)	**0.28** (0.28)	**85**
GS Verify	0.38 (0.39)	0.02 (0.01)	28
Multiplexer			
GS	0.97 (0.05)	0.87 (0.01)	79
GE	0.95 (0.06)	**0.88** (0.04)	56
GS Verify	**0.98** (0.04)	0.49 (0.05)	**87**
Mastermind			
GS	**0.91** (0.04)	0.88 (0.01)	**18**
GE	0.90 (0.03)	**0.89** (0.00)	10
GS Verify	0.90 (0.04)	0.45 (0.09)	13

Table 3.2: A comparison of the results obtained for Grammatical Swarm and Grammatical Evolution extracted from the proof of concept paper [38] and the GS results obtained in this study(GS Verify) over 100 runs across all the problems analysed.

The results provided in the Table above are best analysed by categorizing them into two discussion categories:

- A comparison between the GS implementation presented in the orig-

inal proof of concept paper and that of the GS implemented in the course of this study.

- A comparison between the GE and GS algorithm.

GS versus GS Verify

As can be seen from the results presented in the Table, there are differences in the results obtained by the two implementations on tacking the various problems. It is particularly evident for the mean average finesses, where each of the original GS values are significantly higher than those obtained from the replication. On analysis of the mean best fitnesses results, one can see that the variation between the two is far less, the larges difference obtained being on the Quartic Symbolic Regression problem beating the original GS by `0.07`. On analysis of the total number of successful runs, there are some interesting variations evident. The original GS beats the replicated implementation on both the Santa Fe Ant Trail problem and on the Quartic Symbolic Regression problem. The difference in the number of extra successful solutions obtained by each of the algorithms varies from five (S.F.A.T and Mastermind) to eight (Multiplexer & Symbolic Reg.). A full discussion of these results is presented in the subsection 3.5.1.

GS versus GE

As shown in the results Table, GE outperforms GS on two out of the four problems tackled. The GS performed particularly poor on the Quartic Symbolic Regression problem relative to the GE's performance. Thus, this is a very difficult problem for the GS algorithm. The GS does however, outperform GE on tackling the Mastermind problem. Note, that the information presented here on the GE and GS is is limited and is considered a summary of the study documented in [38]. For a full description of both the implementation details and comparative results of the GS and GE experiments,

refer to that paper.

3.5.1 Discussion

As stated in the results section, there is a difference in the results obtained from both the original GS implementation and the GS verification implementation on tackling the four problem. Every effort was made to ensure that the various parameter and algorithm settings of the replicate was set identical to that of the original version. The most significant variation between the two sets of results can be seen in the mean average fitnesses recorded.

This prompted an investigation into both of implementation and a miscalculation in the average fitnesses outputted in the original implementation was discovered. In the case, of the mean best fitnesses, the variations are too small and are thus deemed insignificant. The total number of successful runs has shown that there is a variation, particularly the second and third problems, where there is a difference of eight full solutions obtained over the one-hundred runs. Interestingly, in this case, each implementation beats each other once.

Taking then mean of the results excluding the miscalculated mean average fitnesses the researcher concludes that the variation is probably due to the *stochastic nature* of the GS algorithm and that overall both GS algorithms have a similar performance level. However, it must be noted that further investigation is warranted, in order to determine the exact cause of the discrepancies in the results.

With regard to the performance comparison between GS and GE, the results suggests that although GS was outperformed by two out of the four problems analyzed it still offers a great deal of potential. This is due to the

fact that the algorithm operated with such a small population of individuals and that it had not been subject to any particular modifications or enhancements.

There is potential for parameter tweaking, natural selection or replacement operators, crossover, introduction of a variable-length approach, etc. Research into parameter tweaking and modification of the basic PSO algorithm itself has proved to be very successful as the algorithm tends to be sensitive even to slight adjustments. Since the PSO is at the heart-beat of Grammatical Swarm, there is good reason to suggest that performing further investigations into improving GS using some combination of the aforementioned enhancement suggestions will strengthen the GS algorithm and possibly yield a significant performance improvement.

One of primary studies of thesis documents the results of an investigation conducted into the effects of one such enhancement. Chapter 5 describes experiments performed into the effects of modifying the PSO particles fixed-length vectors so that the length constraint is removed. Thus, producing a Variable-Length Grammatical Swarm, where vector elements can increase or decrease in size over simulation time.

Chapter 4

Pseudo-Random Number Generator Investigations In Grammatical Swarm

4.1 Introduction

This primary purpose of this Chapter is to document the results of an empirical investigation conducted into the performance effects of using two different quality Pseudo-Random Numbers Generators (PRNGs) on the GS algorithm. The following paragraphs give the breakdown of the Chapters content.

The first Section introduces PRNGs and it explores the significance of random numbers in the EC field and more specifically their significance in GS. This Section also describes the two PRNGs that are used in the experiment. The first of these PRNGs is the system supplied `rand()` and it is considered a poor quality PRNG. The second PRNG is a Mersenne Twister (MT) implementation which is referred to by its class header name `eorng.h`. The

MT is considered a leading PRNG algorithm across the computer science community.

Section 4.3 documents the experimental setup details. This Section is introduced with a discussion on previous PRNG studies conducted on other EAs. Furthermore, the Chapter is concluded with a discussion of the experimental findings which is presented in Section 4.4.

4.2 Pseudo-Random Number Generators

Evolutionary Computation techniques are dependent on the use of *random numbers* as randomness is an important criterion for the successful operation for each of the algorithms in this field. Not only are random numbers responsible for the *scattering* of the populations representations throughout the conceptual landscape but they are also required for various types of updates and/or modifications to the population throughout the entire simulation. In algorithms that are based on GA principles, random numbers are required to apply *reproduction operators* such as stochastic selection, recombination and mutation (See Section 2.1.1). The GS algorithm is no different as it too relies on the use of random numbers. In the PSO learning algorithm of the GS, random numbers are central to the *velocity* update of particle representations.

Researchers and practitioners in the scientific community generally use two primary techniques to generated random numbers, namely, hardware devices and software algorithms. Hardware random number generators operate by extracting random properties from a physical process such as radio interference or thermal noise to produce a range of numbers. Since hardware random number generators tend to be quite slow, expensive and suffer from bias(in that certain numbers maybe outputted more frequently than others)

they are deemed impractical for more common applications and their use is typically confined to specific tasks where a high degree of randomness is necessary such as cryptography, security keys, generating credit card numbers, etc. The second generation technique uses software algorithms to produce random numbers. Such algorithms that are ran on a *deterministic* computer are called Pseudo-Random Number Generators (PRNG). A PRNG by definition cannot produce true random numbers; they can only attempt to approximate the properties of true randomness. John Von Neumann, a founder of the modern computer once stated that

> "*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin* [54]."

This remark emphasizes that it is not a trivial task to produce true random numbers using arithmetical methods. All PRNGs need to be initiated with some value called a seed and a sequence of random numbers is generated from this point on i.e. using the same seed value will produce the same string of numbers regardless. Depending on the memory constraints of a computer the sequence of numbers will eventually begin to repeat itself. However, PRNGs have advanced over the years and today they are capable of generating strings of numbers that are almost indistinguishable from true random numbers and are capable of passing various statistical tests for *randomness*. These statical tests have made it possible to categorise a variety of different PRNG according to their quality. The following Section introduces two different quality PRNGs. It provides a discussion based on material from the available literature on the various reasons why each is either deemed a strong or a weak quality PRNG.

4.2.1 The C/C++ System-Supplied PRNG - `rand()`

As previously stated, this Chapter presents an investigation into the effects of two different PRNGs in the implementation of the Grammatical Swarm

algorithm. This section introduces the first of these, which will be referred to to as `rand()`, which is the standard system-supplied C/C++ algorithm for generating pseudorandom numbers. An identical version of the algorithm is also available on most UNIX based systems as the standard PRNG but in this case it is implemented by calling the `random()` function.

Although, the `rand()` algorithm has been adopted as the standard for the popular C/C++ programming languages and UNIX based systems it is inadequate for most scientific purposes. This fact is highlighted throughout the literature. In a book dedicated to numeric computation methods [45], entitled, *"Numerical Recipes in C: The Art of Scientific Computing"*, the author advises its readers to be very suspicious of system-supplied pseudo random number generators.

The following excerpt from this book emphasizes this point by implying that there is a substantial quantity of scientific papers that may be deemed invalid because researchers may have used bad system-supplied PRNG in their experimentation:

> *"If all scientific papers whose results are in doubt because of bad rand()s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist."*

The author stresses that the period of the `rand()` algorithm is often very large and this can be disastrous in many circumstances.

A paper by Joel Heinrich [11] shows how `rand()` fails the simple maximum spacing test which all popular PRNGs pass. Heinrich stated that the famously flawed `randu` generator generator also failed this test and that the `rand()` algorithm should not be used on any platform. The following pro-

vides a synopsis of the `rand()` algorithms implementation details used in the GS experiments conducted.

Implementation Synopsis

To use the `rand()` PRNG in C/C++, it is necessary to include the `<cstdlib>` or for older (C) compilers the `<stdlib.h>` library header in the program file. These libraries contain various functions that are used to produce random numbers, namely the `rand()` and `srand()`. The `rand()` function returns a random number in the range [0, `RAND_MAX`], where the integer `RAND_MAX` is specified by the system as it varies depending on the compiler; it generally is `32767`. Like all PRNGs the `rand()` algorithm needs to be initiated with an arbitrary *seed*. This is implemented by calling the `srand()` function. The `srand()` function takes the seed as its argument e.g `srand(seed)`. The seed is generally set to a value such as the system clock or some interchangeable values. Of course, the `rand()` function does not, generate *true* random numbers but a sequence of numbers. The sequence starts at the value (position) specified by the selected seed and the sequence will eventually repeat(when `RAND_MAX` is reached). If the same seed is selected then the same sequence of numbers will be generated.

The following code shows the how the `rand()` PRNG is implemented in the C/C++ programming language. Note that this is extracted from the code used in the experimental setup of this study. The seed that is used, line `17`, returns the system time in integer format e.g. `1125416205`. The code details a function `randomNum(double low, double high)` which take the `high` and `low` arguments that determine the range of numbers which will be outputted.

```
1: #include <stdlib.h>
17: srand((unsigned)time(NULL));
23: double randomNum(double low, double high)
24: {
25:  double range=((high-1)-low)+1;
26:  return low+double(range*rand()/(RAND_MAX + 1.0));
27: }
```

4.2.2 An Implementation of the Mersenne Twister PRNG -eoRng

The *Mersenne Twister* [30] is a relatively new PRNG algorithm, introduced in 1998 by Matsumoto and Nishimura. The algorithm was developed as a stringent and efficient method for producing uniform pseudo-random numbers. Until the introduction of the algorithm there was a limited selection of PRNGs that were capable of meeting the demands of the scientific community where there are many disciplines that frequently require an efficient algorithm to produce high quality random numbers. Most of the older PRNGs were inadequate for valid scientific experimentation primarily because they contained flaws. The paper presented by [44] almost ten years before the introduction of the MT, highlights the deficiencies in many of these older PRNGs, which were published in books and even became standards for popular programming languages.

The strength of the MT algorithm can be largely attributed to its impressively large period and equidistribution properties - *"...the algorithm provides a super astronomical period of* $2^{19937} - 1$ *and 623-dimensional equidistribution up to 32-bit accuracy, while using a working area of only 624 words."* It is a successor of the Twisted Generalised Feedback Shift Register (TGFSR)[27, 28] , modified to provide 1) an *incomplete array* which

allows for the provision of the Mersenne-prime[1] period and 2) an efficient algorithm for testing primitivity called the *inversive-decimation* method.

The following provides a summary of the PRNG algorithms key strengths:

- **Long Period** - It has an extremely long period $2^{19937} - 1$ in comparison to other popular generators and has a very large *k-distribution* property.
- **Efficiency** - It does not require large amounts of memory as only consumes 624 words of 32 bits.
- **Speed** - It is very fast as it avoids using multiplication and division being about four times faster than the C/C++ standard `rand()` [29].
- **Statistically Tested** - It has passed statistical test including the stringent diehard tests by George Marsaglia [26] and the load test by Stefan Wegenkittl [8].

The MT algorithm was originally presented in the ANSI C programming language. This study uses a modified implementation of that algorithm to perform a comparative analysis between the MT and rand() PRNGs. This modified implementation is provided in the form of a persistent C++ class in a header file, of the same name, eoRng.h[2]. This library file was developed by Maarten Keijzer. The following paragraph provides a synopsis of its implementation.

[1] A *Mersenne-prime* number is of the form $M_n = 2_n - 1$ where n is a prime number. Mersenne numbers are named after a 17th century French mathematician *Martin Mersenne*. They are the largest known prime numbers today.

[2] Available from:
`http://cvs.sourceforge.net/viewcvs.py/eodev/paradisEO/utils/eoRNG.h`

Implementation Synopsis

The eoRng.h PRNG library must be included at the top of the code file (see line 1 in code segments below) where the pseudo-random numbers are to be generated. The first step in the implementation is to create an object of type eoRng. The constructor of the class takes as its argument a seed(See line number 28 in the code below.) In the case of this implementation an eoRng object called rng is created and the seed is an unsigned long value of exactly *32-bits* in size, identified in the sample code below by the uint32 typedef. Line 27, shows how the seed is declared and assigned a value computed by the performing some arithmetic and a bit shifting operation on the current time and the executed programs process identification number(PID), respectively.

```
1:  #include<eorng.h>
18: typedef unsigned long uint32;
27: uint32 seed = time(NULL) ^ (getpid() << 16);
28: eoRng rng(seed);       //initial seed
42: rng.reseed(seed);      //reseed
```

The various member functions are called depending on the specific function needed. In the case of this PRNG study, random() and uniform() are the primary functions called. The random function returns an random integer between zero and a value passed as an argument to random. An example of this PRNGs use in the implementation is given in sample code below. It shows the rng eoRng class instance invoking the random member function which takes the maximum value of a particle C_{Max} (See Section 3.2.2) as an argument.

Another another member function called flip() is used in some of the experiments presented in the following Chapter, which as its name suggests is used to simulate an instance of a coin flip.

```
for(int i;i<ndim;i++)
{
p[i].current.push_back(rng.random(cmax));
p[i].next ...
}
```

4.3 Experimental Settings

4.3.1 Experiment: The Effects of Two different Quality PRNGs on GS

This experiment examines the performance effects of using two alternative PRNGs on the GS algorithm. The alternative PRNGs were presented in the previous section, namely, the `eorng` MT implementation and the system `rand()`. As explained, these are two different quality algorithms with the MT being much more stringent than that of the standard `rand()`. This experiment intends to determine if the performance of the GS is *effected* by the choice of PRNG.

PRNG investigations have previously been conducted on a variety of EC algorithms. Various studies have illustrated that the performance of evolutionary algorithms can be affected by the choice of PRNG. The research conducted by Daida et al. [6] demonstrates that unexpected improvements were found on diverse performances measures when a weak PRNG was used, improvements could be as substantial as 800%. Foster and Meysenburg [32] also produced some relevant work. They investigated the performance effect of using PRNGs of various quality on a simple GA [31] and they later extended that work, by performing a study [32] on a GP algorithm. Their findings indicated that in very few instances, a weak PRNG resulted in slight performance improvements however Foster and Meysenburg found no

Setting	Value
Population Size	30
Population Size	30
Particle Vector Length	100
Iterations	1000
VMAX	±255
C_{Min}	0
C_{Max}	255
Wrapping Operator	On
Max Wraps Allowed	10
Number of Runs	100

Table 4.1: The experimental settings for both GS implementations on the PRNG investigation.

evidence of stronger GA performance with good PRNGs. Thus there is no evidence to support the notion that improved PRNG quality caused improved GP performance. Erick Cantu-Paz [4] conducted a set of experiments to show the effect of PRNGs on a simple GA and to identify the components that are most affected by the PRNG. The result of the experiments show that the PRNG used to initialize the population is vital while the impact of the PRNG used as input to other operations is relatively insignificant to performance. In an effort to ensure that experiment results are interpreted consistently, Cantu-Paz states that it is advisable to use the best PRNG available.

This PRNG investigation, was conduced by tackling each of the four problems using two implementations of the GS algorithm. These two implementations were identical in every aspect apart from the type of PRNG used. The first implementation used an instance of the `eoRng` and the second used the `rand()`. This allowed for a comparative study between the two alternative PRNGs. The results of the experiment are described with the aid of plotted graphs of the fitness, showing both the mean average and mean best

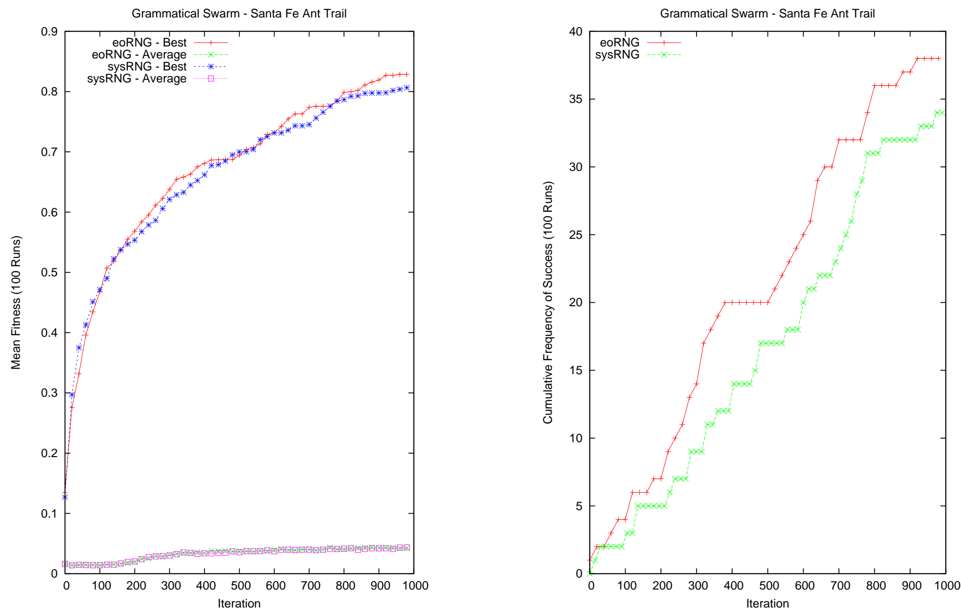


Figure 4.1: Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which showing the mean fitness(left) and the cumulative frequency of success(right) on tackling the Santa Fe Ant Trail problem.

fitness and cumulative frequency of success attained by the particles in the swarm over the course of the simulation. Table 4.1 shows the various parameter and population initialization settings chosen for each implementation.

Santa Fe Ant Trail

The plots presented above in Figure 4.1, represent the average fitness values sampled over 100 simulation runs by both GS implementations (`eorng` and `rand()`), showing the mean fitness(left) and the cumulative frequency of success(right) on tackling the Santa Fe Ant Trail problem. The mean fitness samples, both best and average are almost identical although the `eorng` implementation obtaining a slightly higher *best* fitness value. The difference is far too small to be considered significant. However, the plot

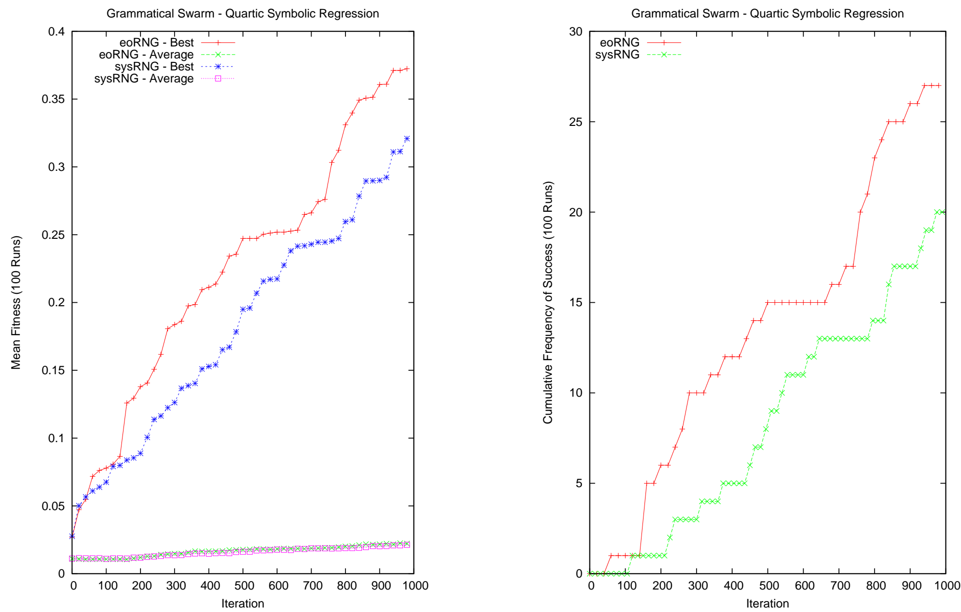


Figure 4.2: Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of sauces(right) on tackling the Quartic Symbolic Regression problem.

of the cumulative frequency of success shows a definite improvement, it is evident that the `eorng` implementation solved the Santa Fe Ant problem more times than that of the alternative `rand()` implementation.

Quartic Symbolic Regression

The results obtained from the Quartic Symbolic Regression problem tackled are presented in the two plots above in Figure 4.2. It is evident from the results shown in both the best fitness(left) and the cumulative frequency of success plots(right) that the GS implementation based on the superior quality `eorng` PRNG outperforms the GS implementation that uses the alternative, `rand()` PRNG. There is a significant performance improvement in terms of the total number of successful solutions obtained by the `eorng` GS implementation. It outperforms the weaker PRNG by a total of `8`. This

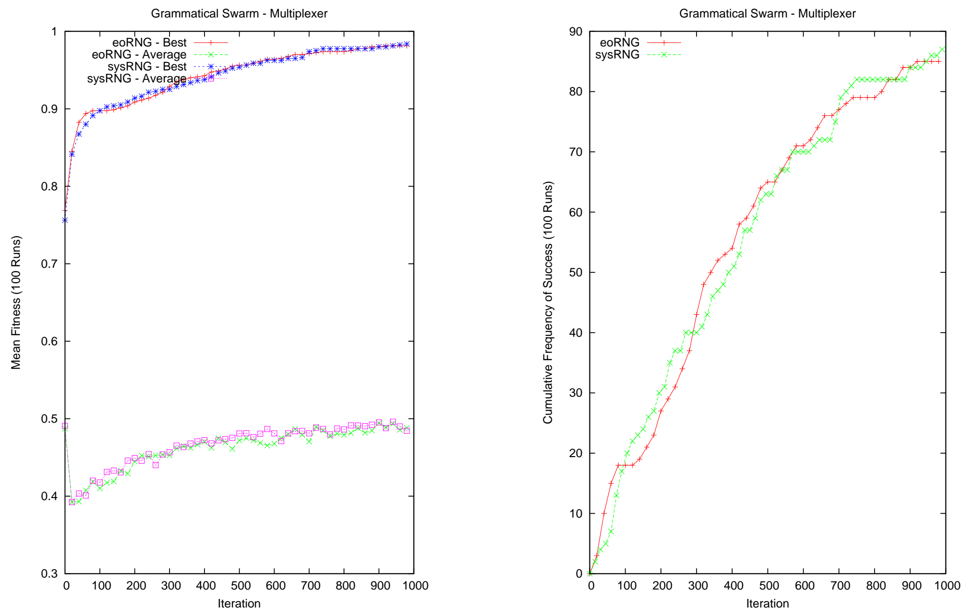


Figure 4.3: Plot of the results achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of success(right) on tackling the 3 Multiplexer problem.

indicates that for this type of problem domain the choice of PRNG could have a significant impact on the performance of the algorithm.

3 Multiplexer

Figure 4.4 shows the mean fitness(left) and cumulative frequency of success(right) plots for the two alternative GS implementations on tackling the 3 Multiplexer problem indicate that there is no difference between the two implementations. The mean best and average fitness and the total number of successful runs are all equal. Thus, one can conclude that for this type of problem the choice of PRNG does not in any way influence the performance of the GS algorithm on this problem domain.

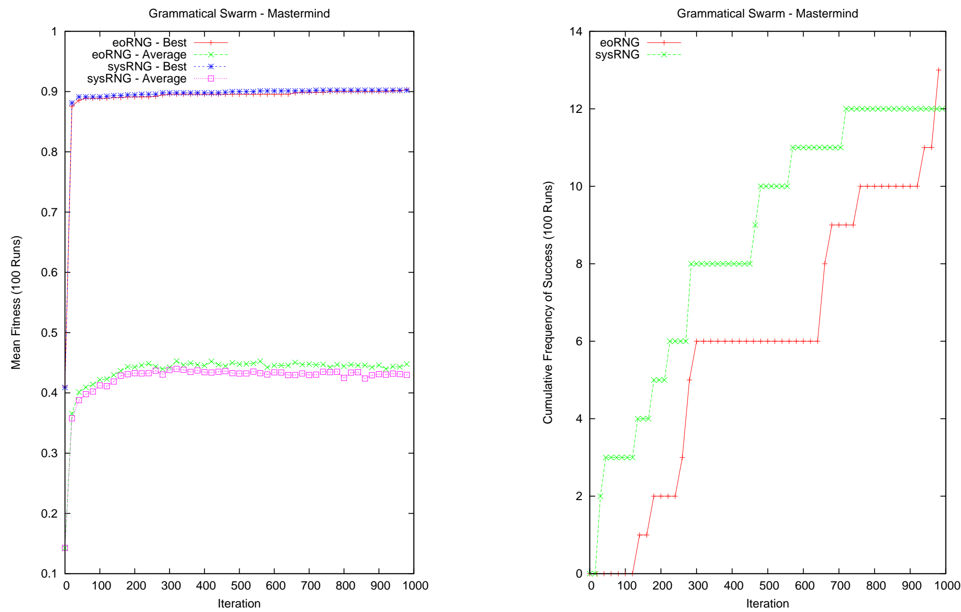


Figure 4.4: Plot of the fitness achieved by the two GS implementations, `eoRng` and `rand()` which shows the mean fitness(left) and the cumulative frequency of success(right) on tackling the Mastermind problem.

Mastermind

Figure 4.4 shows the plots of the results obtained for this problem. The plot of the best fitness(left) shows that there is a tie between both of the GS implementations. However, there is a difference between the two implementations in the cumulative frequency plot(right). This shows that the `eorng` GS implementation does solve the mastermind problem more than times, although the difference is very slight i.e it only outperforming the `rand` implementation one time more out a hundred runs. Needless to say that the difference is deemed insignificant and thus one can conclude that in this type of problem the choice of PRNG does not effect the overall performance of the GS algorithm.

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
S.F.A.T			
`rand()`	0.81 (0.19)	**0.04** (0.01)	34
`eorng`	**0.83** (0.19)	**0.04** (0.01)	**38**
Q.S.R			
`rand()`	0.32 (0.34)	**0.02** (0.01)	20
`eorng`	**0.38** (0.39)	**0.02** (0.01)	**28**
Multiplexer			
`rand()`	**0.98** (0.04)	**0.49** (0.05)	**87**
`eorng`	**0.98** (0.04)	**0.49** (0.05)	**87**
Mastermind			
`rand()`	**0.90** (0.04)	0.43 (0.12)	12
`eorng`	**0.90** (0.04)	**0.45** (0.09)	**13**

Table 4.2: A summary of the results for each of the problems tackled by both PRNGs, showing the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem.

4.4 Discussion

Table 4.2 provides a summary and comparison of the results obtained for two GS implementations, each using a different quality PRNG on tackling the various problem domains. It shows that in terms of the total number of successful solutions obtained for each of the problem domains over one hundred runs, the GS implemented with the `eorng` PRNG outperforms the `rand()` PRNG on three of the four problems, although there is only a difference of `1` run on the Mastermind problem. The performance is matched on the 3 Multiplexer problem, with each implementation attaining a `87` runs each. It is particularly evident from the Quartic Symbolic Regression problem that the two GS implementations are affected by the choice of PRNG. With regards to the mean average fitness results, there is *no* significant performance difference evident. The mean best fitness performance results show that there is, again a significant difference on the Quartic Symbolic Regression problem. A sight difference between the two implementations is

also evident on the Santa Fe Ant Trail. However, the other two problems achieve identical mean best fitnesses regardless of the choice of PRNG.

In conclusion, this investigation had the objective of determining each of the following:

- If the choice of PRNG influences the performance of the GS algorithm and to what extent.
- If a GS implemented with a high quality PRNG (such as the `eorng`) outperforms a GS implemented with a poor quality PRNG(such as the system `rand()`).
- If the case of a performance difference, is the performance difference evident across all problem domains or is it specific to certain types of problems.

It is evident from the results that the choice of PRNG does, in fact, influence the performance of the GS algorithm on certain problems, with the stronger PRNG implementation outperforming the weaker implementation. The extent of the performance difference varies depending on the type of problem, for half of the problems there was only a very slight difference (Mastermind) to no difference being evident at all (Multiplexer). Note that the `eorng` will be adopted as the PRNG of choice for each of the experiments conducted in Chapter 5.

Chapter 5

Variable-Length Grammatical Swarm

5.1 Introduction

This Chapter presents a series of investigations into the effects of modifying the vector-lengths representations in the PSO *search* component of the GS algorithm. The primary objective of the Chapter is to (1) examine the effects of increasing particle representation sizes and (2) introduction a new implementation of the GS algorithm, where the vector-length constraint is removed which is referred to as the Variable-Length GS algorithm.

The following gives the structure of the Chapter:

- Section 5.2, documents an investigation conducted into the effects of modifying the canonical fixed-length vector representations of GS. This experiment involves increasing the vector-length of the population of *particle* representations from 100 codons to 200 codons i.e. double its original size. The results are presented in the form of a comparative study between the original GS (100) and the larger 200 codon GS.

- Section 5.3, introduces the Variable-Length GS, giving an overview of the algorithm's initialisation details and the four different variable-length particle updated *strategies* developed. This Section presents the following proof of concept Variable-Length GS experiments:
  - Experiment A: A Variable-Length GS Initialised with `100` Codons
  - Experiment B: A Variable-Length GS Initialised with `200` Codons
  - Experiment C: A Comparative Analysis of the `100` and `200` Codon Variable-Length GS.
- The final experiment analyses the `evolution` of the size of the variable-length particle representations throughout simulation on tackling the various problem domains presented in Chapter 3. This experiment was conducted in an effort to gain further insight into the extent of the variable-length *particle* change and more specifically to determine if the GS suffers from *bloat*.
- The final Section presents an extensive discussion based on the results obtained from each of the experiments in the Chapter.

5.2 Particle Size Investigations in Fixed-Length GS

The particles in the GS algorithm [38] are represented as *fixed-length* vectors. A full description of the implementation details concerning the fixed-length aspect of the particles used in GS is provided in Section 3.2.4. In this section we present the results of an experiment which was conducted in order to determine the performance effect of *increasing* the size of the particles' vectors or in other words *increasing* the amount of instructions that describe a solution i.e. contained in the *genome*.

In the conventional GS, the length of a particle was *bounded* to a hard-length constraint of 100 elements, in the experiment documented here, this value is doubled by initializing each particle in the swarm so that they contain vectors of 200 dimensions (or codons). Adjusting the size of the vectors in this way, means that during the mapping process there are exactly twice as many codons in the genome.

This experiment aims to determine if an increased amount of codons will assist the swarm in finding a solution with greater speed or perhaps finding an ever better solution or both. As the quantity of codons, is effectively doubled, there is an increased possibility that a large number of codons will be redundant in the mapping of `terminals` to `non-terminals` as a solution can *potentially* be produced with a small number of codons, particularly with the use of the wrapping operator. However, the fact that the redundant codons are considered *introns* (i.e. they may be switched on in subsequent iterations) and giving that number of codons constituting the interons in this modified GS will typically be larger than those in the conventional 100 codon implementation, may have a positive impact on the performance of the modified implementation. For example, in the mapping of a genome (particle) which contains 100 codons (elements), where the first 50 elements are used to derive `terminals` from `non-terminal` to results in the construction a full solution. The remaining 50 unused codons are considered introns. An objective of this experiment is to determine if carrying extra genetic information (in this case double i.e. 100 to 200), which will probably be unused for the majority of the simulation, does give the swarm an advantage.

Setting	Value
Population Size	30
Particle Vector-Length	200
Iterations	1000
VMAX	+-255
C_{Min}	0
C_{Max}	255
Wrapping Operator	On
Max Wraps Allowed	10
Number of Runs	100
PRNG	`eoRng`

Table 5.1: The experimental settings for the Fixed-Length GS particle vector size investigation.

5.2.1 Results

Two variants of the GS are used in this experiment:

- The conventional Fixed-Length GS initialised with all the parameters as described by O'Neill and Brabazon in [38]. This GS implementation is identified in the results by the following, *GS*, key.
- The same conventional Fixed-Length GS implementation as above but in this case each of the particle vector representations is initialised such that it now contains `100` more elements (codons) i.e. a total dimension size of `200` for each of the **`current`**, **`next`** and **`best`** vectors. The additional elements are also populated with random values real-values in the range $[C_{Min}, C_{Max}]$. This GS implementation is identified in the results below by the, *GS(200)*, key.

These two GS implementations were used to tackle each of the benchmark problems (See Section 3.4.1). The mean *average* and mean best particle scores were recorded. The results compare the mean over a total of 100 runs which is presented in the form of graphs in the following subsections under the headings of the corresponding problem tackled. Table 5.1 summarises

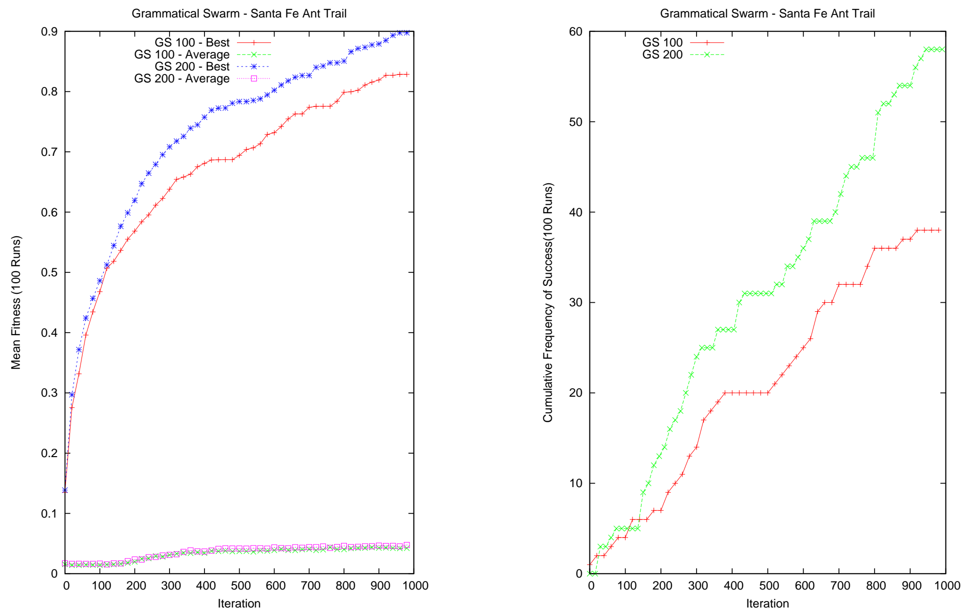


Figure 5.1: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the `100` codon and `200` codon GS implementations on tackling the Santa Fe Ant Trail problem.

the implementation details used in this experimental setup.

Santa Fe Ant Trail

The Santa Fe Ant trail problem was tackled by both the original 100 dimension particle representation, *GS* and the 200 dimension representation, GS(200). Both graphs show that there is a significant difference between the two implementations for this type of problem. Although, the mean fitness graph(left) shows that the mean average fitness difference between both implementations is very slight, the *GS(200)* implementation achieved a much higher *mean best* fitness, scoring a maximum of `0.90` at the last iteration compared to a mean best fitness of `0.83` for the *GS*. The improved performance is much more evident, in the cumulative frequency of success plot(right). There is a very significant improvement with the 200 codon im-

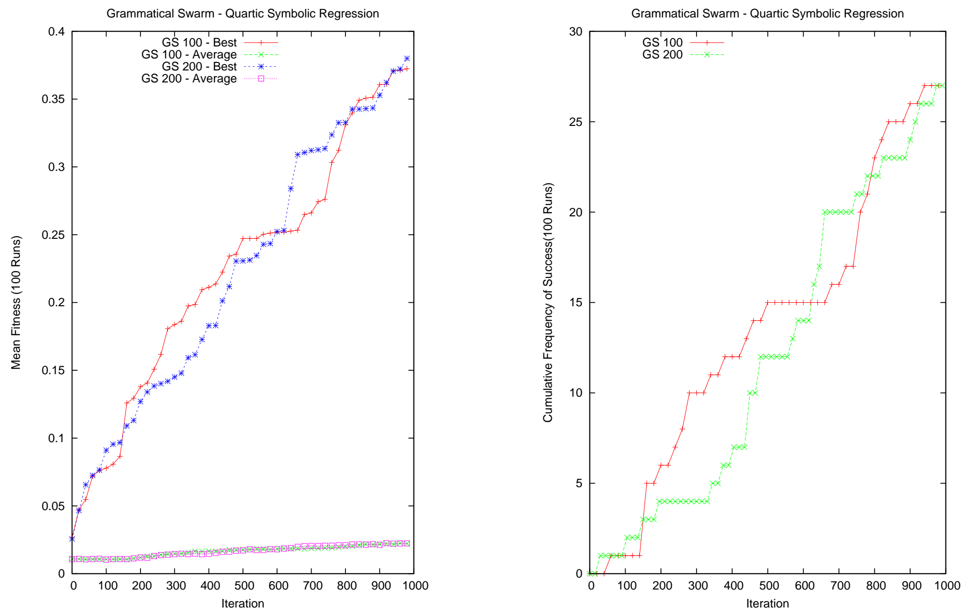


Figure 5.2: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the 100 codon and 200 codon GS implementations on tackling the Quartic Symbolic Regression problem.

plementation which solves the ant problem 20 times more than that of the *GS* implementation.

Quartic Symbolic Regression

An instance of the Quartic Symbolic Regression problem was tackled by both of the GS implementations, for 100 runs each. As can be seen from the two plots presented above in Figure 5.2, the mean best and mean average fitness(left) and the cumulative frequency(right) of success results are almost identical. The only difference being the cumulative frequency plot for the *GS*, obtaining one more successful run than the *GS(200) implementation.* However, the difference is too small to be of any significance. Overall, the results show that the there is *no* gain in performance achieved when using an increased particle vector representation.

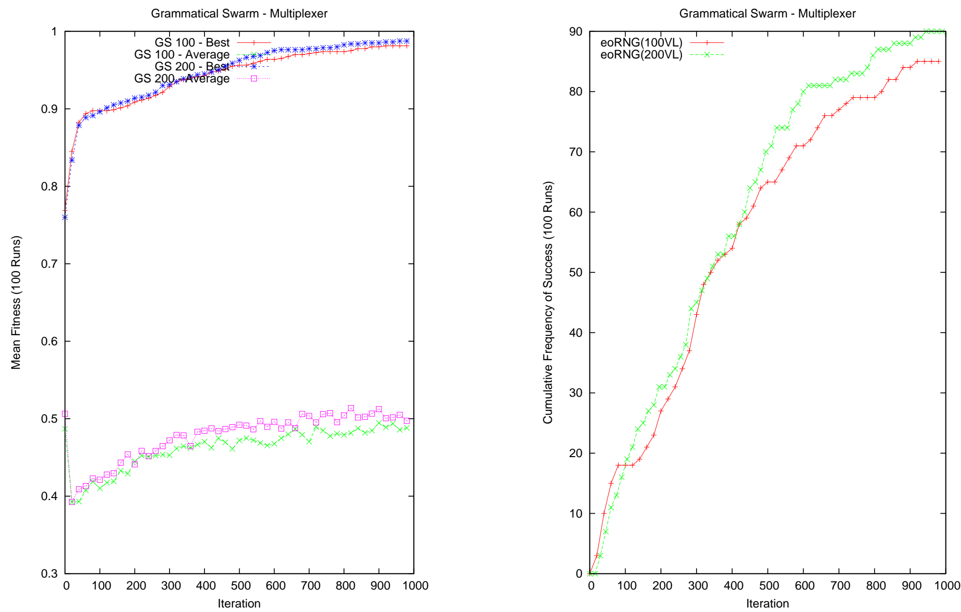


Figure 5.3: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the 100 codon and 200 codon GS implementations on tackling the 3 Multiplexer problem.

The 3 Multiplexer

The multiplexer problem is the third problem tackled in this experiment. The mean fitness(left) and cumulative frequency of success(right) plots are shown above in Figure 5.3. As can be seen the mean best and average fitness differences are very slight. The mean best for *GS(200)* achieved a very slight improvement with it recorded a value of 0.99 compared to 0.98.The same is true for the mean average fitness values recorded, with the 100 vector, *GS*, sored 0.49 compared to the 200 vector implementation, *GS(200)*, 0.5. The cumulative frequency of success plot shows that the *GS(200)* is evidently the stronger algorithm in this problem domain; it found the target solution 5 times more that that of the 100 vector implementation.

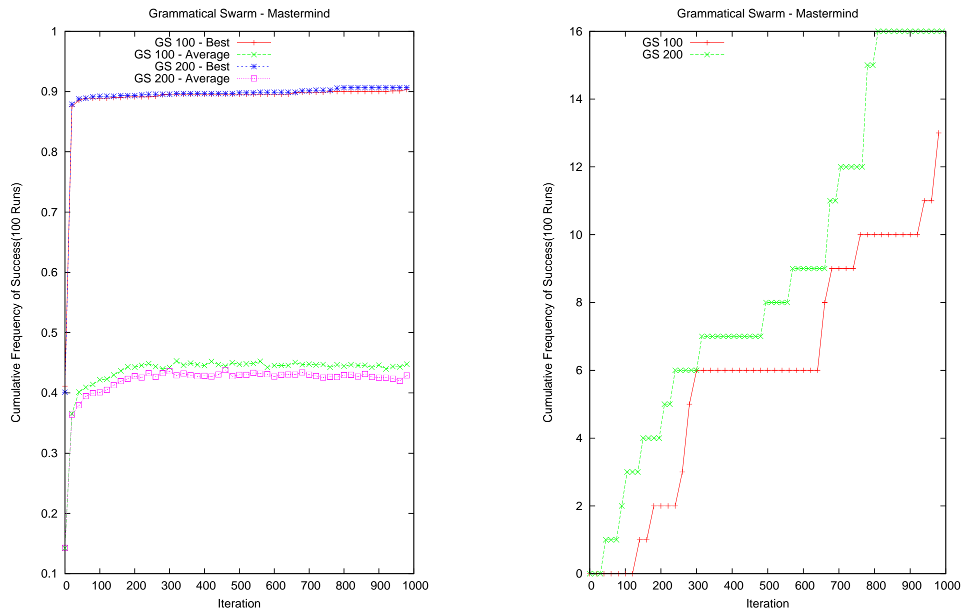


Figure 5.4: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the 100 codon and 200 codon GS implementations on tackling the Mastermind problem.

Mastermind

An instance of the mastermind problem was the fourth problem tackled. The first plot(right) in Figure 5.4 shows the mean best and average fitnesses shows that the plots are identical, apart from a very slight improvement by the 200 codon *GS(200)*implementation. As can be seen from the cumulative frequency of success plot(left), there is an improvement, with the 200 codon *GS(200)* solving the problem a total of 3 times more. Note that the mastermind is a particularly difficult problem for the both the GS and the GE(See Section 2.4) algorithms, therefore any slight improvement such as that obtained here, could in fact, be very promising.

5.3 Introducing the Variable-Length Grammatical Swarm Representation

This following introduces the novel variable-length implementation of the Grammatical Swarm algorithm which will be referred to as the Variable-Length GS. In contrast to the GS algorithm presented in Chapter 3 and Chapter 4 this implementation adopts variable-length particle vectors. Although the elements were bounded in length in this canonical GS, not all elements were necessarily used to construct a program during the mapping process, and as such the programs generated were variable in size. This study aims to determine if a variable-length particle representation can outperform the fixed-length GS representation. The following will present the Variable-Length GS implementation details as well as four different implementation strategies. The results of each of the strategy implementations on tackling the four problem domains are also presented subsequently.

5.3.1 Initialisation

For each particle in the swarm a random number is generated in the range $[dim_{Min}, dim_{Max}]$, where dim_{Min} is set to 1 and dim_{Max} is set to the desired maximum number of codons. This random number determines the length of the particles vector. For example, if the number 40 is randomly generated for a particle in the swarm. This number is used to determine the vector length of that particle. Thus, in this case, it constitutes a vector of 40 dimensions in length. After each of the thirty particles have been created with variable length vectors, they are then initialised. The particles current vectors are initialised with random numbers in the range $[C_{Mim}, C_{Max}]$ and similarly the velocity vectors are also populated with random numbers, however they are bound to values drawn from ± 255.

Four different approaches to a variable-length particle swarm algorithm were investigated in this study.

5.3.2 Variable-Length GS Strategies

Strategy I - Each particle in the swarm is compared to the global best particle (g_{best}) to determine if there is a difference between the length of the current particles vector and the length of the g_{best} vector. If there is no difference between the vector sizes then an update is not required and the algorithm simply moves on and compares the next particle to g_{best}. However, when there is a difference between the vector lengths, the particle is either extended or truncated. If the current particles, p_i vector length is shorter than the length of g_{best}, dimensions are added to the particles vector extending it so that it is now equivalent in length to that of g_{best}. The particle's new dimensions contain values which are copied directly from g_best. For example, if g_{best} is a vector containing fifty dimensions and the current particle has been extended from forty five to fifty dimensions then the values contained in the last `5` dimensions (`46-50`) of g_best are copied into the five new dimensions of the current particle. If the particle has a greater number of dimensions than the g_{best} particle, then any the extra dimensions are simply truncated so that both g_{best} and the current particle have equivalent vector lengths.

Strategy II - This strategy is similar to Strategy I, the only difference is the method in which the new dimensions are copied. In the first strategy, when the current particle, x_i is extended the particle's new dimensions are populated by values which are copied directly from g_best. In Strategy II, values are not copied from g_best instead random numbers are generated in the range $[C_{Min}, C_{Max}]$ and these values are copied into each of x_i's new dimensions.

Strategy III - The third strategy involves the use of probabilities. Given a specified probability, the length of the particle is either increased or decreased. A maximum of one dimension can only be changed at a time i.e. either a dimension is either added or removed from the current particle, x_i. If x_i is longer than g_{best} then the last dimension of $x_i$is discarded. If x_i is shorter than g_{best} then x_i is increased by adding an extra dimension. In this situation the new dimension takes the value of a random number in the range [C_{Min}, C_{Max}].

Strategy IV - The fourth strategy involves the generation of a random number to determine the number of dimensions that will be added to or removed from the current particle, x_i. If the length of $x_i$is shorter than the length of g_{best} the difference, *dif* between the length of g_{best} and the length of x_i is calculated. Then a random whole number is generated in the range [0, *dif*]. The result of this calculation is then used to determine how many dimensions will be truncated from x_i. A similar strategy is applied when the length of x_i is smaller than the length of g_{best}. However, in this case the random number generated is used to determine the number of dimensions that x_i will be extended by. After x_i is extended, each of these extended dimensions are then populated with random numbers generated in the range [C_{Min}, C_{Max}].

A strategy is not applied ever time it was possible to modify x_i, instead, applying a strategy is determined by the outcome of a certain probability function i.e. the outcome of this function is used to determine if a strategy is to be applied to x_i. In our implementation, a probability of 0.5 was selected. Therefore 50% of the time a specified strategy is applied and 50% of the time x_i is not modified at all.

Setting	Value
Population Size	30
Particle Vector-Length	Variable
Particle Vector-Length Initialisation	[1,100]
Sensible Initialisation	Off
Iterations	1000
VMAX	+-255
C_{Min}	0
C_{Max}	255
Wrapping Operator	On
Max Wraps Allowed	10
Number of Runs	100
PRNG	`eoRng`

Table 5.2: The experimental settings for the Variable-Length GS particle vector size investigation.

5.4 Proof of Concept Experiments and Results

5.4.1 Experiment A: A Variable-Length GS Initialised with `100` Codons

In order to determine the feasibility of generating solutions using a GS algorithm with variable-length particles, four Variable-Length GS implementations were produced and the performance of each implementation was measured. Each of these four implementations were identical in every aspect except for the strategies that they employed to update the swarm particles. These strategies were presented in Section 5.3.2 above. The graphs in the following subsections present the results obtained by the four implementations on tackling the various problem domains. Each of the implementations are referred to in the following subsections according to the strategy(Strategy I - Strategy IV) which it employed. Table 5.2 provides a summary of the settings used in the in each of the Variable-Length GS algorithms.

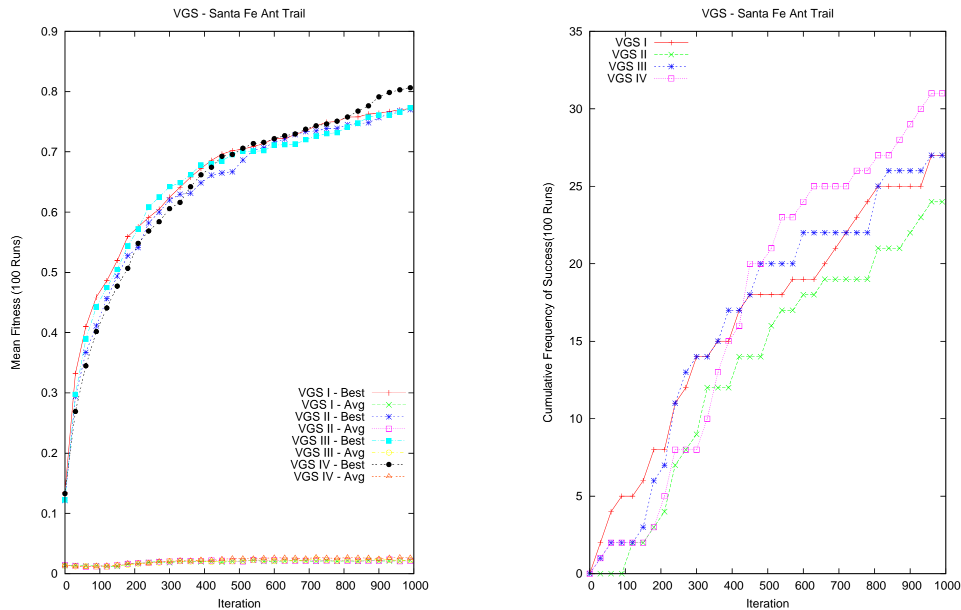


Figure 5.5: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Santa Fe Ant Trail problem.

5.5 Results

Santa Fe Ant Trail

The four Variable-Length GS implementations each tackled an instance of the Santa Fe Ant Problem. The results of these are presented in the mean fitness and cumulative frequency of success plots in Figure 5.1. The first plot shows that the most successful implementation is the one which employed, *Strategy IV*, scoring a total an average best fitness of `0.80` and an average best fitness of `0.03`. The worst performance in terms of mean fitness was *Strategy II*. The cumulative frequency of success plot show that the *Strategy IV* is also the best at finding correct solution; it did so a total of `31` times. The weakest performance was *Strategy II* as it found a successful solution 7 times less than that of *Strategy IV*.

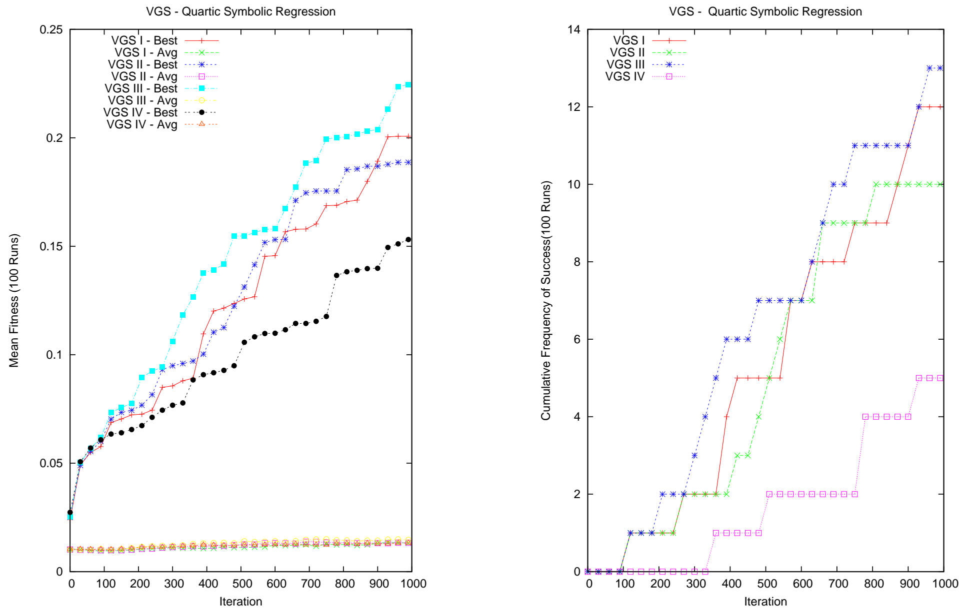


Figure 5.6: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Quartic Symbolic Regression problem.

Quartic Symbolic Regression

The Quadratic Symbolic Regression problem was also tackled by the four GS implementations. It is evident from the two plots presented above in Figure 5.6 which show the mean fitness and cumulative frequency of success that there the different strategy implementations have a significant effect on the performance of the Variable-Length GS algorithm for this type of problem. The third strategy is the winner in terms of both mean best and successful runs. It obtained a total of mean best fitness of `0.23`and it found the solution `13` times. The implementation that employed *Strategy IV* was by far the weakest, scoring only `0.25` for its mean best fitness and finding the only finding a full solution `5` times. The mean average fitness was the same for all four implementations, at `0.01`.

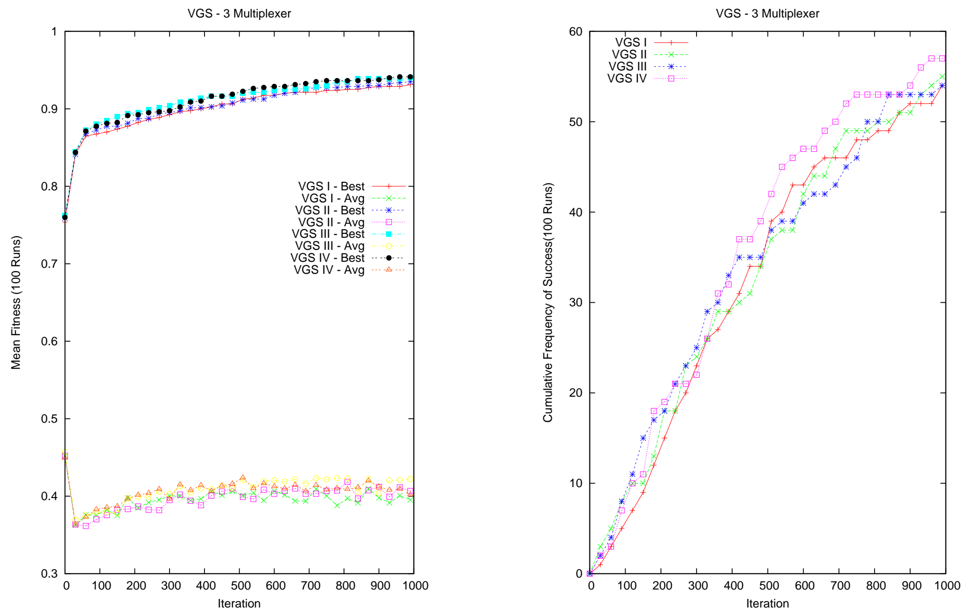


Figure 5.7: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the 3 Multiplexer problem.

The 3 Multiplexer

The multiplexer problem was the third problem tackled by the GS implementations. The results (Figure 5.7) obtained from this problem suggest that there is little difference in the various implementations. This is particularly evident form the mean best fitness plot, where the three of the strategies converge at the same point, `0.94` and the fourth, *Strategy I* converges just below it scoring a mean best fitness of `0.93`. The cumulative frequency of success plot, also shows that there is little difference with the weakest strategy finding the solution `54` times in comparison to the *Strategy IV*, which finds it three times more.

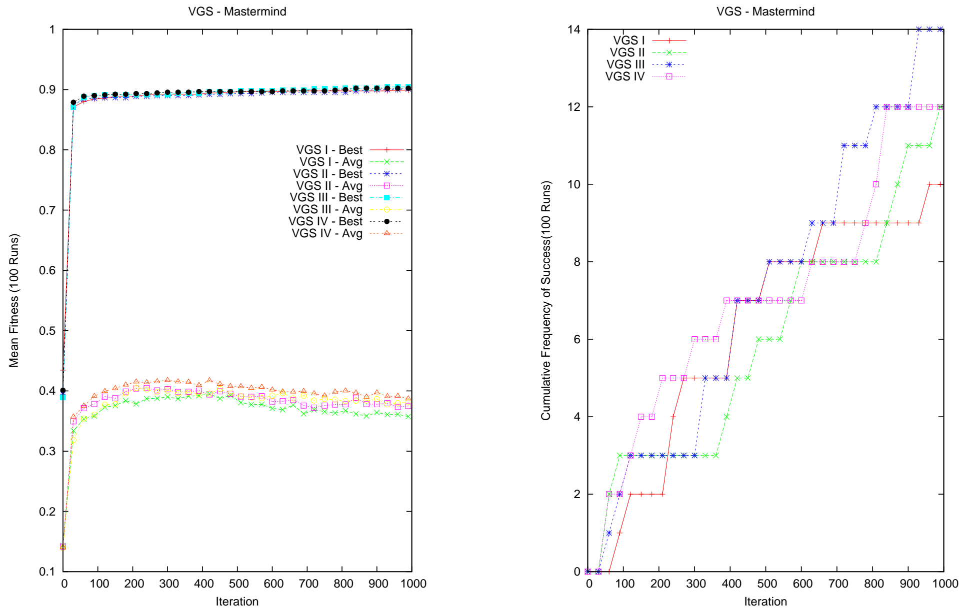


Figure 5.8: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `100 codons` on tackling the Mastermind problem.

Mastermind

An instance of the mastermind problem was tackled by the four GS implementations, in order to further verify the feasibility of producing solutions in the form of computer programs using a variable-length GS particle representation. The results from this problem are shown in Table 5.8. The first plot of the mean *best* and *average* fitnesses shows that the plots are almost identical, apart from a very slight disimprovement in terms of mean best fitness runs by the *Strategy I* implementation. The cumulative frequency of success plot shows that the difference between the implementations is also very slight. The best performance was achieved by *Strategy III*, with a total of `14` successful runs while the first strategy only solved the problem `10` times.

5.5.1 Experiment B: A Variable-Length GS Initialised with 200 Codons

Section 5.2 presented the results of a study conducted on the performance effect of an increased particle size in the Fixed-Length GS. In that study, the size of the particles' vectors was increased to 200 dimensions. The swarm particles in the canonical GS, presented in [38], were 100 dimensions in size, therefore that study was conducted with a particle vector size(or the number of codons) that was effectively doubled. As can be seen from the the results of that experiment a significant improvement can be obtained by implementing a GS with a *larger* particle size representation. These improved results prompted a similar study on the Variable-Length GS implementation. The following subsections presents the results of that study.

Unfortunately, it is not possible to conduct the exact same particle size study on the Variable-Length GS. The reason for this is obvious, because each variable-length particle in the search space does not contain a *fixed* number of dimensions. Instead, the same initialization procedure as described in Section 5.3.1 is used but in this case, during particle initiation the random numbers are drawn from the range $[1, 200]$ as opposed to the previous range of $[1, 100]$. Each time a particle is initialised, its length must be set, as each particle is variable in length a number is chosen randomly and this number determines the length of the particle. Increasing the range (from 100 to 200) that the random numbers are drawn from, effectively increases (doubles) the potential vector length of each particle in the swarm. Thus the 200 particle size or 200 codon Variable-Length experiment presented here is setup in a very similar method to the bounded implementation study of 5.2 in an effort to ensure a relatively fair comparison of experiment results.

This study is conducted similarly to the the 100 codon Variable-Length

Setting	Value
Population Size	30
Particle Vector-Length	Variable
Particle Vector-Length Initialisation	[1, 200]
Sensible Initialisation	Off
Iterations	1000
VMAX	+-255
C_{Min}	0
C_{Max}	255
Wrapping Operator	On
Max Wraps Allowed	10
Number of Runs	100
PRNG	`eoRng`

Table 5.3: The experimental settings for the Variable-Length GS particle vector size investigation for the 200 codon implementation.

GS implementation presented in Experiment A 5.4.1. Again, *four* different Variable-Length GS algorithm implementations are produced, one for each of the four strategies. In the case of the particles are initialised as described in the previous paragraph i.e. the random number generator function responsible for determining the number of dimensions that a particle may contain is modified so that numbers are now drawn from an increased range. The four implementations are used to tackle the various problems, `100` times for each problem and for each strategy. The results are provided in the following subsections, under the heading of the respective problem tackled. Table 5.3 provides a summary of the settings used for each of the implementations.

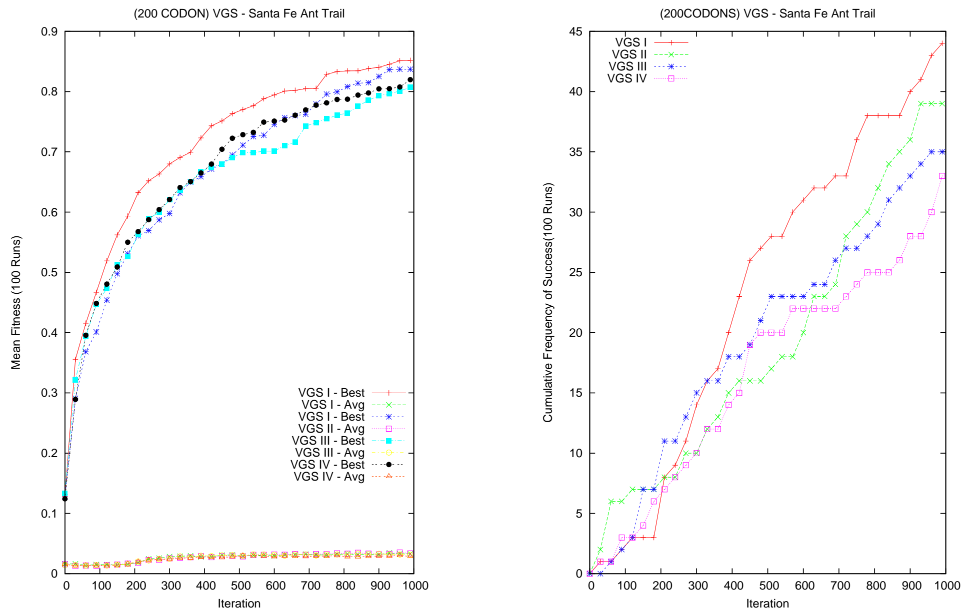


Figure 5.9: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Santa Fe Ant Trail problem.

5.6 Results

Santa Fe Ant Trail

The first problem tackled by the 200 codon implementation is the Santa Fe Ant Trail. As can be seen from Figure 5.9 in both mean and best fitness plots(left), *Strategy I* achieves the highest fitness score and the *Strategy IV* implementation is the weakest. The cumulative frequency plots confirm this, as the first strategy is obviously the strongest, solving the problem a total of 45 times and *Strategy IV* is also the weakest in terms of successful runs, finding a full solution only 33 times.

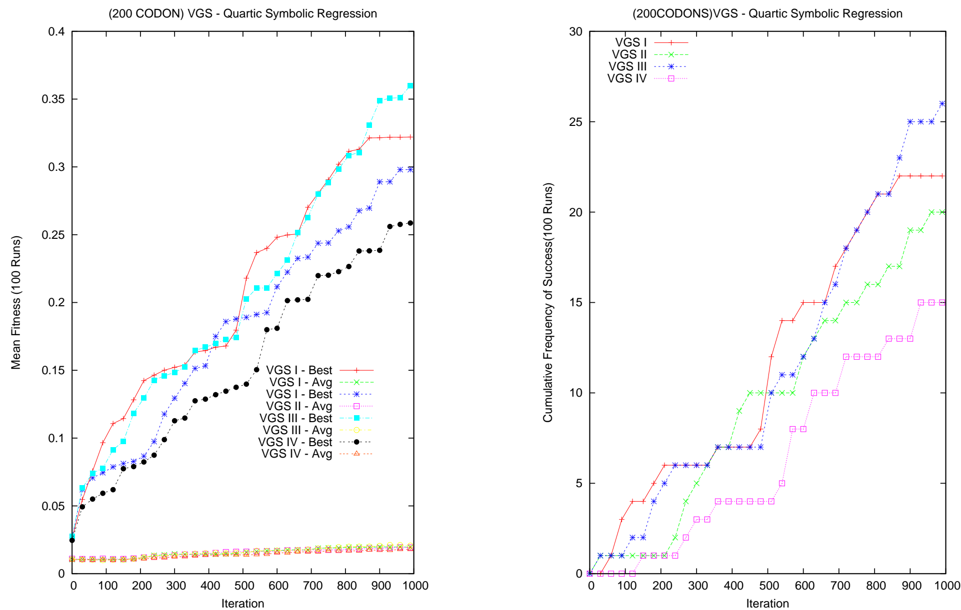


Figure 5.10: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Quartic Symbolic Regression problem.

Quartic Symbolic Regression

Each of the four increased codon length, Variable-Length GS implementations tackled an instance of the quartic symbolic regression problem. The third strategy implementation was by far the strongest. This is *evident* from Figure 5.10 in both the mean and average fitness plot(left) and the cumulative frequency of success plot(right). This implementation solves the problem `26` times and achieves an mean best fitness of *0.36*. As in the Santa Fe Ant problem, the implementation that uses *Strategy IV* performs the worst, solving the problem just `15` times, with a mean best of just `0.26`.

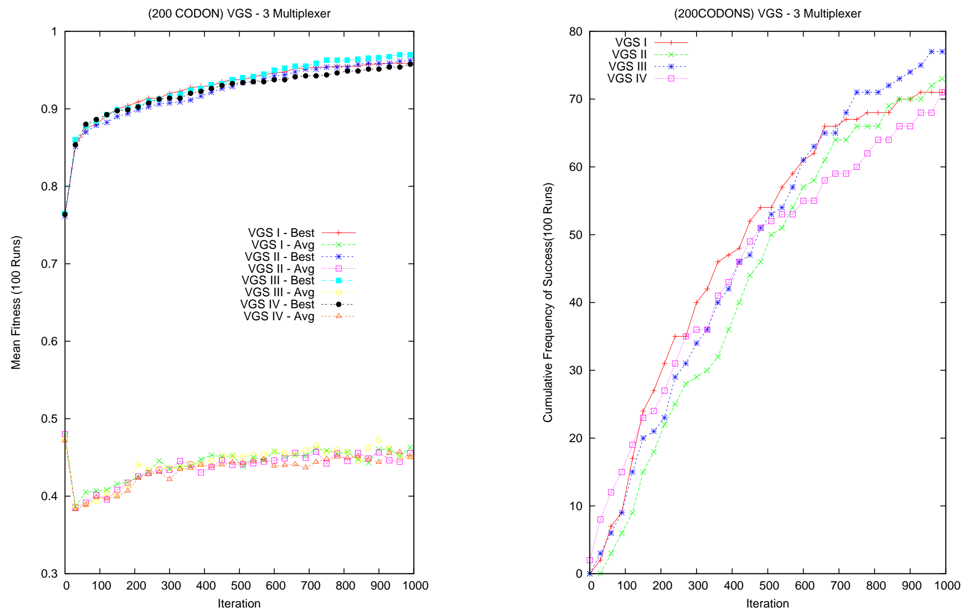


Figure 5.11: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with 200 `codons` on tackling the 3 Multiplexer problem.

The 3 Multiplexer

The third problem tackled was the 3 Multiplexer. Figure 5.11 shows that *Strategy III* is the most successful for this type of problem. It is particularly evident in the cumulative frequency of success plot(right) where it evidently outperforms the other strategies, finding the target multiplexer 77 times compared to the weakest *Strategy I* which solves the problem 6 times less. Although this its success is not as evident in the mean fitness plot(left) it is still achieves the highest score.

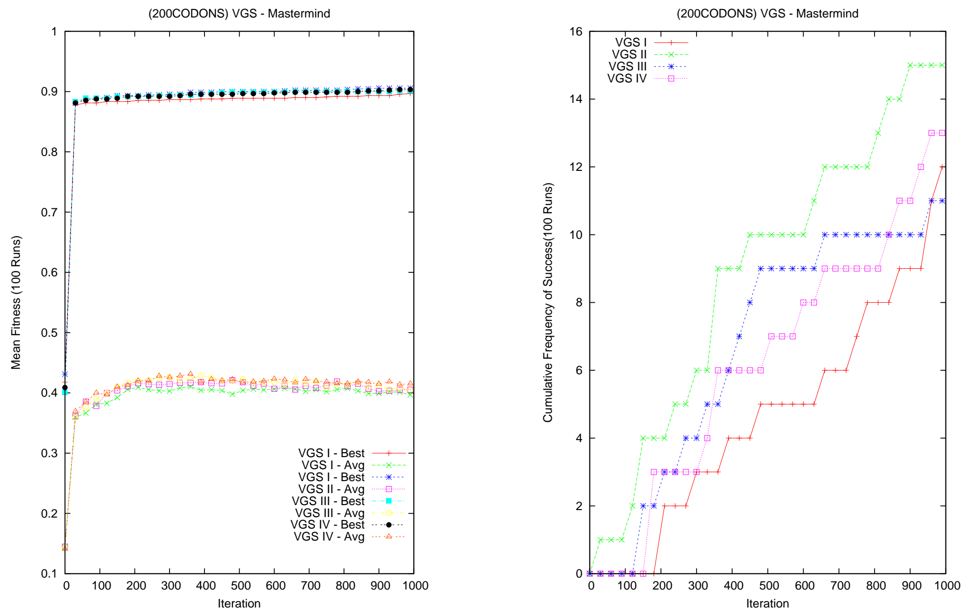


Figure 5.12: Plot of the *Mean Fitness*(left) and *Cumulative Frequency of Success*(right) for the four Variable-Length GS strategies initialised with `200 codons` on tackling the Mastermind problem.

Mastermind

An instance of the mastermind if tackled by each of the four implementations. Figure 5.12 shows both mean best fitness(left) and cumulative of success(right) plots of the results. It is evident that *Strategy II* is the most successful. It achieves a total of `15` successful runs and it also narrowly beats the competing implementations in terms of mean best fitness.

5.6.1 Experiment C: A Comparative Analysis of the 100 and 200 Codon Variable-Length GS

The objective of this Section is to provide a comparison between the two Variable-Length Grammatical Swarm implementations presented in the previous experiments in Section 5.4.1 and Section 5.3. The results of these experiments are summarised in Table 5.4 on page 98. The Table shows the *Mean Best*, *Mean Average* and the total number of *Successful Runs* attained by the corresponding algorithm implementation. The four strategies are compared for both the 100 and 200 codon implementations under the headings of the various problem domains.

The results show that there is no clear winner in terms of the most successful strategy. When looking at the mean best fitness and total number of successful runs, this variation in performance becomes evident with Strategy I being the most successful on the Santa Fe Ant Trail, Strategy II on the Mastermind and Strategy III wins on both the Quartic Symbolic Regression and 3 Multiplexer problems. Thus, it is difficult to form a definite conclusion with regard to the strategies, however it should be recognised that the different strategies can significantly affect the performance of the Variable-Length GS depending on the type of problem domain tackled.

One can form a more definite conclusion with regard to the different codon length initialisation. It is evident for the results that the 200 codon Variable-Length GS achieves higher performance levels when compared to the 100 codon implementation. The 200 codon implementation in the clear winner it attained a higher mean best fitness and it solved the problems the most times across all four problems. Thus, it is recommended to initialise the Variable-Length GS to 200 condon for maximum performance.

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
Santa Fe Ant			
(100 Codon) I	0.77 (0.17)	0.02 (0.02)	27
(200 Codon) I	**0.86** (0.19)	**0.03** (0.02)	**45**
(100 Codon) II	0.77 (0.19)	0.02 (0.02)	24
(200 Codon) II	0.84 (0.19)	**0.03** (0.02)	40
(100 Codon) II	0.78 (0.20)	0.02 (0.02)	27
(200 Codon) III	0.80 (0.19)	**0.03** (0.02)	35
(100 Codon) IV	0.80 (0.19)	**0.03** (0.02)	31
(200 Codon) IV	0.81 (0.18)	**0.03** (0.02)	33
Q.S.R			
(100 Codon) I	0.20 (0.30)	0.01 (0.01)	12
(200 Codon) I	0.32 (0.37)	**0.02** (0.07)	22
(100 Codon) II	0.19 (0.28)	0.01 (0.01)	10
(200 Codon) II	0.29 (0.36)	**0.02** (0.01)	20
(100 Codon) III	0.23 (0.31)	0.01 (0.01)	13
(200 Codon) III	**0.36** (0.38)	**0.02** (0.01)	**26**
(100 Codon) IV	0.16 (0.20)	0.01 (0.01)	5
(200 Codon) IV	0.26 (0.32)	**0.02** (0.01)	15
Multiplexer			
(100 Codon) I	0.93 (0.08)	0.40 (0.11)	54
(200 Codon) I	0.95 (0.07)	**0.46** (0.08)	71
(100 Codon) II	0.94 (0.08)	0.40 (0.10)	55
(200 Codon) II	0.96 (0.07)	0.45 (0.08)	73
(100 Codon) III	0.94 (0.07)	0.40 (0.09)	54
(200 Codon) III	**0.97** (0.06)	**0.46** (0.08)	**77**
(100 Codon) IV	0.94 (0.07)	0.40 (0.09)	57
(200 Codon) IV	0.95 (0.07)	**0.46** (0.09)	72
Mastermind			
(100 Codon) I	0.90 (0.04)	0.36 (0.15)	10
(200 Codon) I	0.89 (0.06)	0.39 (0.14)	12
(100 Codon) II	0.90 (0.04)	0.36 (0.14)	12
(200 Codon) II	**0.91** (0.04)	0.40 (0.13)	**15**
(100 Codon) III	0.90 (0.04)	0.38 (0.14)	14
(200 Codon) III	0.90 (0.03)	0.40 (0.13)	11
(100 Codon) IV	0.90 (0.04)	0.38 (0.15)	12
(200 Codon) IV	0.90 (0.04)	**0.41** (0.11)	13

Table 5.4: A comparison of the results obtained by the 200 codon and 100 codon particle initialised Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four strategies.

5.7 Variable-Length GS Particle Size Evolution

In many population-based algorithms that use variable-length representations experience a problem where the size of the solution representation increases in length during the evolutionary process. This increase can be very large and it is referred to as *bloat* or *fluff* [23]. Bloat increases the structural complexity of the algorithm and results in an is a very inefficient algorithm especially in terms of computer memory resources. Many studies have been conducted on the bloating of representations such as [22, 23, 21, 36, 53, 51, 52, 25]. There are many general explanation for bloat which can be generally applied to any *progressive search technique*. Langdon defines bloat as

> *"The increase in program's size from one generation to the next while the performance of programs within the population is essentially the same as in previous generations."*

Bloat is very common in GP, therefore in an effort to determine if bloat occurs in the Variable-Length GS an investigation was performed which explores the evolution of particle size in the Variable-Length GS algorithm and the results of this experiment are present in this section.

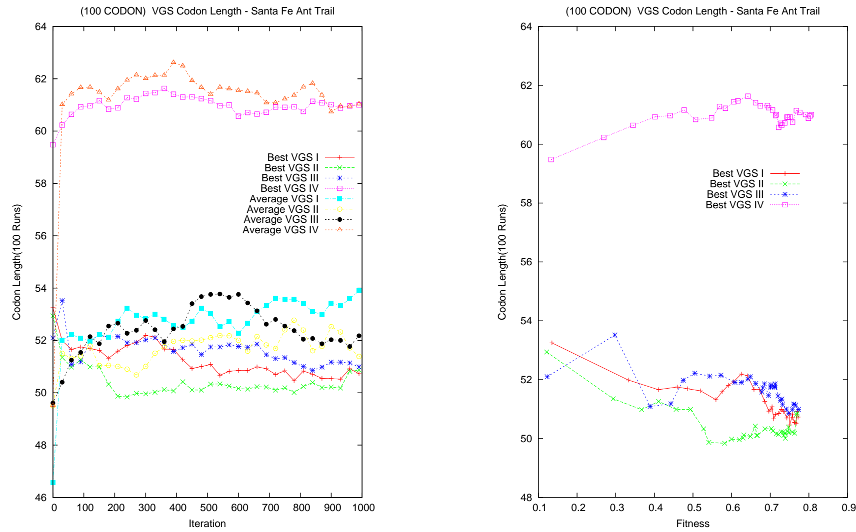
The investigation aims to determine each of the following:

- Does bloat occur in the Variable-Length GS algorithm and if so is it specific to a particular problem domain or a set of problem domains.
- Do particles converge at a certain size and if so, is there a particular particle size or range that a particle achieves a good fitness.
- Is there a common pattern of particle size change throughout the `1000` iterations.

The experiment was designed such that at each iteration, the vector size of the best and average particles were recorded in the problems tackled in the experiment described in the Section 5.4.1 i.e for the 100 Codon GS Initialisation. That is, the particle size (codon length) results presented here were extracted from Experiment A. The results are presented below in the form of two plots, the first shows the plot of the size of the mean best and average particle vector (or codon) size at each iteration in the experiment. The second plot shows the best particle size of the mean best particle at a certain fitness. E.g. a particle with a mean best fitness of 0.6 has in of 40 dimensions in size. A table accompanies each of the four problems in an effort to aid in the analysis of the particle size and mean fitness comparison. In each of these tables the codon lengths are shown delimited by parenthesis', where the value under the *Mean Best Fit.* column is the *Mean* g_{best} *Codon Length* and the values which fall under the heading of *Mean Avg Fit.* represent the *Mean Average Codon Length* recorded.

5.7.1 Results

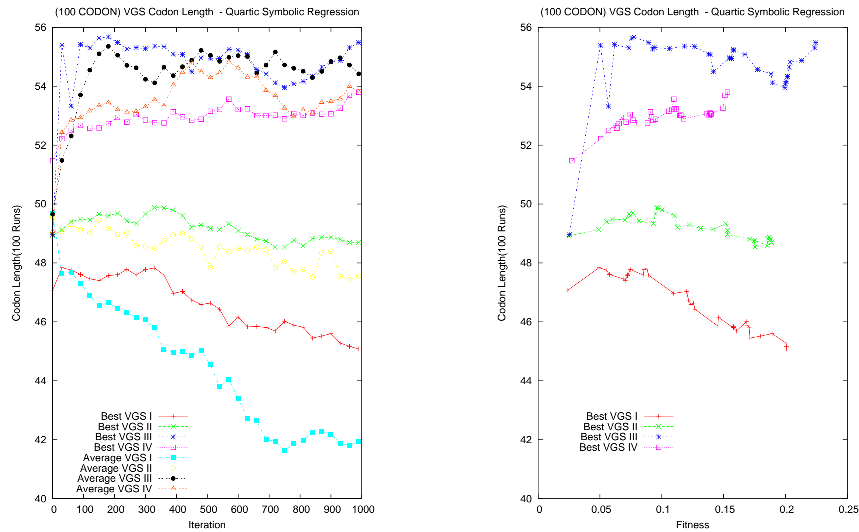
Santa Fe Ant Trail



	Mean Best Fit. (Cod. Len.)	Mean Avg. Fit. (Cod. Len.)	Successful Runs
Santa Fe Ant			
I	0.77 (50)	**0.02** (54)	27
II	0.76 (51)	**0.02** (51)	24
III	0.78 (51)	**0.02** (51)	27
IV	**0.80** (61)	**0.02** (61)	**31**

The two plots show that the codon length of the mean g_{best} and average particle is in the range [50, 61]. This suggest that bloat is not an issue for this type of problem. The *Strategy IV* implementation which solved the problem the most times and had the achieved the highest fitness out of all strategies, had a particle length of 61. This may employ that larger particles perform better. It is also interesting to note that the particle size seems to fluctuate frequently throughout the iterations(albeit small changes).

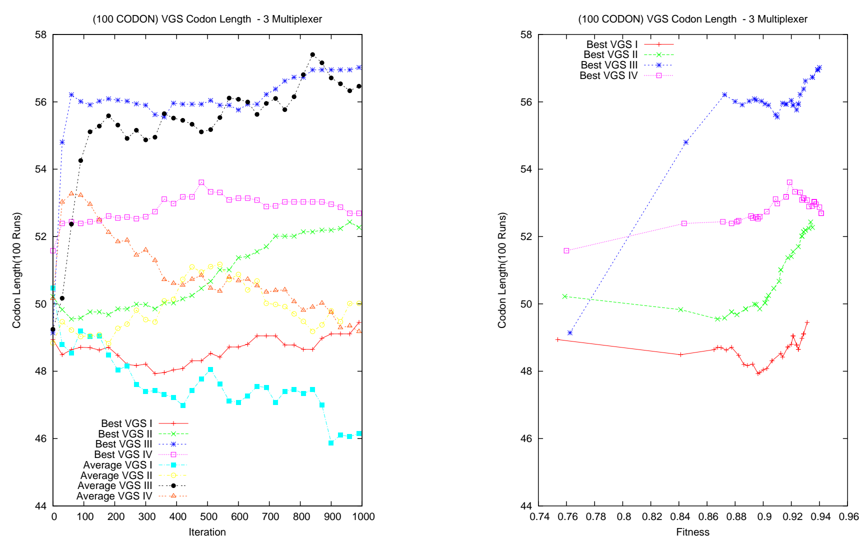
Quartic Symbolic Regression



	Mean Best Fit. (Cod. Len.)	Mean Avg. Fit. (Cod. Len.)	Successful Runs
Q.S.R			
I	0.20 (45)	**0.01** (42)	12
II	0.19 (49)	**0.01** (48)	10
III	**0.23** (55)	**0.01** (54)	**13**
IV	0.16 (54)	**0.01** (54)	5

The results from the Quartic Symbolic Regression problem show that the particle length never go beyond the range [42, 55] for mean best and average fitness. In the Santa Fe Ant Trail problem the particles with the most codons score the highest fitness both in terms of mean best and average and total number of successful runs, this is also the case in this problem. However, the lowest fitness has also has a relatively high particle size. Therefore, the large the particles, do not necessarily mean attain high a fitness nor outperform the particles with fewer codons.

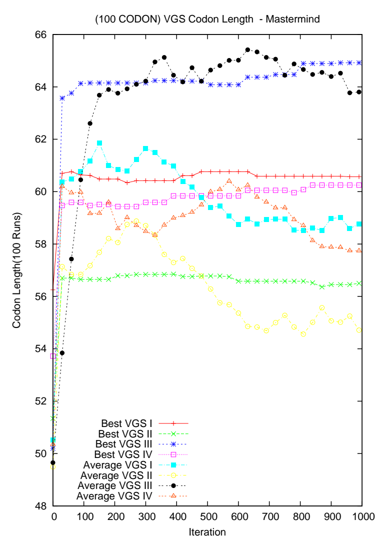
The 3 Multiplexer

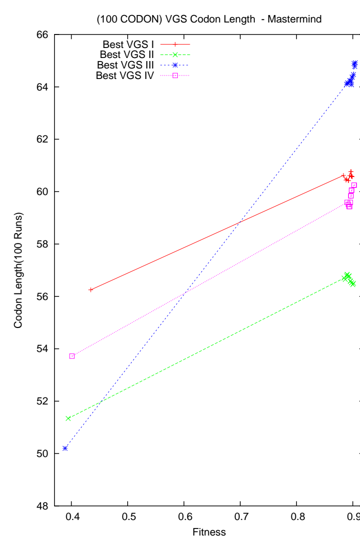


	Mean Best Fit. (Cod. Len.)	Mean Avg. Fit. (Cod. Len.)	Successful Runs
Multiplexer			
I	0.93 (49)	**0.40** (46)	54
II	**0.94** (52)	**0.40** (49)	55
III	**0.94** (57)	**0.40** (56)	54
IV	**0.94** (53)	**0.40** (49)	**57**

As can be seen in the plots and table, the 3 multiplexer problem result are too similar to draw a conclusion with regards to the codon length, however the evolution of the codon size shows that the particles tend to grow rapidly at the start until a certain fitness point is reached, then a high period of fluctuation is evident for the remainder of the iterations.

Mastermind





	Mean Best Fit. (Cod. Len.)	Mean Avg. Fit. (Cod. Len.)	Successful Runs
Mastermind			
I	0.89 (61)	0.36 (59)	10
II	**0.90** (57)	0.36 (55)	12
III	**0.90** (65)	**0.38** (64)	**14**
IV	**0.90** (60)	**0.38** (58)	12

The particle size for the mastermind is presented in the plots above along with a table which gives additional information. The plots are somewhat similar the ones attained on tackling the multiplexer problem. Again we see that bloat is certainly not an issue, with the maximum number of codons being 65. Slight fluctuation of particle size throughout the course of the simulation is also evident here.

5.8 Discussion

Each of the experiments are discusses below under its corresponding experiment heading with the aid of *result summary* Tables where deemed appropriate.

5.8.1 Fixed-Length Particle Size Investigation

The first experiment which is documented in Section 5.2, performed an investigation into the effects of modifying the canonical fixed-length particle representation GS. The modification referred to is in the form of an increased number of codons in the particle representations. The canonical GS presented in the [38] and in the experiments described in the previous Chapters adopted a GS implementation with particles consisting of *100* codons. This study doubles the size of the particles to *200* codons in an effort to gain a performance improvement. Table 5.5 gives a summary of the results recorded.

It is shown from the table that a significant performance improvement can be obtained by doubling the size of the codons in the fixed-length vector. This improvement is most evident in terms of total number of successful runs and in terms of mean best fitness. A very significant improvement was gained on tackling the Santa Fe Ant Trail problem, beating the smaller 100 codons implementation by a total of 20 runs and the mean best fitness improved by 7%. The differences were not as large on the 3 Multiplexer and Mastermind problems, with no significant improvement in terms of mean fitness, however the number of successful solutions does increase. The Quartic Symbolic Regression showed no significant change. One can say that the 200 codon GS implementation is superior to the 100 codon implementation and it it is suggested that the 200 codon version should be adopted, certainly when tackling problems such as the Santa Fe Ant Trail, 3 Multiplexer and

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
S.F.A.T			
100 Codons	0.83 (0.19)	0.04 (0.01)	38
200 Codons	**0.90** (0.16)	**0.05** (0.01)	**58**
Q.S.R			
100 Codons	0.38 (0.39)	0.02 (0.01)	**28**
200 Codons	0.38 (0.38)	0.02(0.01)	27
Multiplexer			
100 Codons	0.98 (0.04)	0.49 (0.05)	87
200 Codons	**0.99(0.03)**	**0.50** (0.05)	**92**
Mastermind			
100 Codons	0.90 (0.04)	**0.45** (0.09)	13
200 Codons	**0.91** (0.04)	0.43 (0.12)	**16**

Table 5.5: A summary of the results for the Fixed-Length Particle Size Experiments, showing the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* on tackling each of the problem domains.

Mastermind.

5.8.2 Variable-Length GS 100 Codon Implementation

Two different experiments were conducted using the Variable-Length GS algorithm. The first of these adopted a 100 codon initialisation strategy i.e. the maximum number of codons that a particle can take on is 100 or a maximum vector size of 100 dimensions. A summary of the results for each of the four strategies on tackling the various problems is shown in the Table 5.6. It is clear that the different strategies do influence the performance of the GS algorithm. While there is no clear winner across all four problem strategies, III and IV were the most successful overall, with Strategy IV producing best performance on the Santa Fe Ant Trail and Multiplexer problems, while Strategy III had the better performance on the Quartic Symbolic Regression and Mastermind instances.

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
Santa Fe Ant			
I	0.77 (0.17)	0.02 (0.02)	27
II	0.77 (0.19)	0.02 (0.02)	24
II	0.78 (0.20)	0.02 (0.02)	27
IV	**0.80** (0.19)	**0.03** (0.02)	**31**
Q.S.R			
I	0.20 (0.30)	**0.01** (0.01)	12
II	0.19 (0.28)	**0.01** (0.01)	10
III	**0.23** (0.31)	**0.01** (0.01)	**13**
IV	0.16 (0.20)	**0.01** (0.01)	5
Multiplexer			
I	0.93 (0.08)	**0.40** (0.11)	54
II	**0.94** (0.08)	**0.40** (0.10)	55
III	**0.94** (0.07)	**0.40** (0.09)	54
IV	**0.94** (0.07)	**0.40** (0.09)	**57**
Mastermind			
I	**0.90** (0.04)	0.36 (0.15)	10
II	**0.90** (0.04)	0.36 (0.14)	12
III	**0.90** (0.04)	**0.38** (0.14)	**14**
IV	**0.90** (0.04)	**0.38** (0.15)	12

Table 5.6: A summary of the results obtained by the [1, 100] particle initialised Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four Variable-Length GS strategies.

5.8.3 Variable-Length GS 200 Codon Implementation

In the fixed-length particle size investigation documented Section 5.3, an investigation was performed into the performance effects of increasing the particle representation to double its original size. A similar[1] study was performed on the Variable-Length GS and it is documented in Section 5.3. A summary of the results for each of the four Variable-Length GS strategies

[1]The Variable-Length GS uses a different implementation strategy to that of its fixed-length counterpart therefore it was not possible to perform the study such that every aspect of the variable-length implementation was identical to the fixed-length implementation, however an adjustment was made to the Variable-Length GS implementation in an effort to achieve a relatively fair comparison (See Section 5.3)

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
Santa Fe Ant			
I	**0.86** (0.19)	**0.03** (0.02)	**45**
II	0.84 (0.19)	**0.03** (0.02)	40
III	0.80 (0.19)	**0.03** (0.02)	35
IV	0.81 (0.18)	**0.03** (0.02)	33
Q.S.R			
I	0.32 (0.37)	**0.02** (0.07)	22
II	0.29 (0.36)	**0.02** (0.01)	20
III	**0.36** (0.38)	**0.02** (0.01)	**26**
IV	0.26 (0.32)	**0.02** (0.01)	15
Multiplexer			
I	0.95 (0.07)	**0.46** (0.08)	71
II	0.96 (0.07)	0.45 (0.08)	73
III	**0.97** (0.06)	**0.46** (0.08)	**77**
IV	0.95 (0.07)	**0.46** (0.09)	72
Mastermind			
I	0.89 (0.06)	0.39 (0.14)	12
II	**0.91** (0.04)	0.40 (0.13)	**15**
III	0.90 (0.03)	0.40 (0.13)	11
IV	0.90 (0.04)	**0.41** (0.11)	13

Table 5.7: A summary of the results obtained by the $[1, 200]$ particle initialised (200 Codon) Variable-Length GS Experiment. It shows the *Mean Best* and *Average* Fitness' with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four Variable-Length GS strategies.

on tackling the various problems is shown in Table 5.7.

Firstly, it is obvious that there is a significant performance increase to be gained compared to the 100 codon variable-length implementation. Secondly, as is the case in the 100 codon implementation, the choice of strategy does effect the performance of the algorithm. In this 200 codon implementation, there no one strategy wins on all problems. The results vary considerably across each of the problem domains. Strategy III is the most successful as it is the winner on two of tackling both the Quartic Symbolic

Regression and Multiplexer problems. On the Santa Fe Ant Trail problem, Strategy I is most successful and Strategy II wins on tacking an instance of the Mastermind problem.

5.8.4 Variable-Length GS Comparative Analysis

Section 5.6.1 presented a comparison of the two Variable-Length experiments, namely the 100 codon implementation and the 200 codon implementation. As perviously discusssed, the results showed that none of the strategies were completley successful across *all* problem domains, instead there was a large variation in their performances. For instance, a Strategy that performed best on the Santa Fe Ant Trail problem could perform very poor on the Mastermind problem. It is recommended that the type of strategy selected should consider the problem domain that will be tackled. As for the number of codons that the algorithm should be initialised to, it is recommended that the larger 200 codon implementation should be adopted regardless of the type of problem.

5.8.5 The Evolution of Size in the Variable GS

The results from the analysis into the evolution of size in the Variable-Length GS particle representations suggest that the is no evidence of bloat. This is an interesting observation as in many variable-length Evolutionary Algorithms the representations tend to grow over simulation time. However, the results show that the more successful particles tend to be those with the most codons; this is the case in three out of the four problem domains. The *absence* of bloat is a characteristic of the swarms behaviour warrants further investigation.

	Mean Best Fit. (Std.Dev.)	Mean Avg. Fit. (Std.Dev.)	Successful Runs
Santa Fe Ant			
GS (Var)	0.86 (0.19)	0.03 (0.02)	45
GS (Fix)	**0.90** (0.16)	**0.05** (0.01)	**58**
Q.S.R			
GS (Var)	0.36 (0.38)	**0.02** (0.01)	26
GS (Fix)	**0.38** (0.39)	0.02 (0.01)	**28**
Multiplexer			
GS (Var)	0.97 (0.06)	0.46 (0.08)	77
GS (Fix)	**0.99** (0.03)	**0.50** (0.05)	**92**
Mastermind			
GS (Var)	**0.91** (0.04)	0.40 (0.13)	15
GS (Fix)	**0.91** (0.04)	**0.43** (0.12)	**16**

Table 5.8: A comparison of the results obtained by the best Variable-Length GS and best Fixed-Length GS. It shows the *Mean Best* and *Average* Fitness with *Standard Deviations* (delimited with parenthesis') and the total number of *Successful Runs* for each problem tackled by the four strategies.

Table 5.8 presents a final comparison between the best performances of both fixed-length and variable-length representation algorithms, in terms of total number of successful runs, across all strategies and codon implementations. It is shown that the fixed-length implementation achieves better performances as it beats the variable-length counterpart on all problems. The results in this table were attained by the larger `200` on all problems apart from the Quartic Symbolic Regression where its best performance was achieved by with the `200` codon implementation. As such, the recommendation at present would be to adopt the Fixed-Length GS implemented with particle representation of `100` codons.

In summary, the following conclusions can be drawn from the series of investigations examined in this Chapter:

- Doubling the total number of codons in the particle representation in the Fixed-Length Grammatical results in a significant performance

improvement across all problem domains.

- It is possible to generate computer programs to solve a diverse set of benchmark program-generation problems using a variable-length particle swarm algorithm.
- The Variable-Length GS does not suffer from *bloat*.
- The overall best performance achieved on tackling the various problems was achieved using the Fixed-Length GS implemented with particle representation of 200 codons.

Chapter 6

Conclusion

This Chapter consists of two sections; the first gives a brief overview of the thesis highlighting the primary findings and contributions and the second discusses possible research directions that could strengthen the Grammatical Swarm (GS) algorithm by extending and refining the work presented in this study.

6.1 Summary

Chapter 2, explored the scientific background of this thesis. Firstly it introduced the concept of numeric *optimisation* and this was followed by an overview of the Evoutionary Computation (EC) methodology which provided an explanation of the more traditional population-based Evolutionary Algorithms(EAs). This formed the grounding for the introduction of two more resent EC techniques, namely the Grammatical Evolution(GE) and Particle Swarm Optimisation(PSO).

The research conducted in this study is based on Social Programming or more specifically the GS Algorithm which was presented in Chapter 3. GS is a hybrid algorithm consisting of a Particle Swarm learning algorithm coupled to Grammatical Evolution genotype-phenotype mapping to generate

programs or solution in an arbitrary language.

This thesis documented the results of a number of empirical investigations into the GS algorithm. The first of these investigation conducted was presented in Chapter 3. This was a verification experiment, in that it had the objective of reproducing an existing study. Specifically, a GS algorithm was developed so that it was identically in every aspect to that of the original GS presented in that algorithms *proof of concept* paper [38]. The new implementation was constructed based entirely on the information on that paper using the same settings and parameters as the original. The newly implemented algorithm was then evaluated on the same set of test problem. The results showed that there was some variation in the two sets of results with a significant difference between the mean average results recorded. Upon investigation, the deviations were found to be the result of a simple calculation error in the original implementation and thus, both GS implementations were considered valid.

Chapter 4 presented an empirical investigation conducted into the performance effects of using two different quality Pseudo-Random Numbers Generators (PRNG) on the GS algorithm. Firstly, an overview of the significance of random numbers in the EC field and more specifically their significance in GS is given. It highlights the difficulties involved in producing random numbers by deterministic methods. This leads onto a discription of the following two PRNGs which are used in the experiment: (1) the system supplied `rand()` and (2) an implementation of the Mersenne Twister called `eorng`. The `rand()` PRNG is referred in the litrature [11, 45, 44] as being of poor quality while the latter is considered a leading method of producing pseudorandom numbers across the scientific community [26]. Next, the experimental setup and results are presented. Finally, a discussion of those

results is given which concludes that the choice of PRNG does, in fact, influence the performance of the GS algorithm on certain problems. However, for half the problems analysed, the choice of PRNG did not affect the performance in any significant way.

Chapter 5, presented a series of empirical experiments conducted on the GS algorithm. The first of these performed an investigation into the effect of *increasing* the size of the algorithms' particles fixed-length vector representations. In the canonical GS, particles have a hard-length constraint consisting of 100 codons. In this experiment the number of codons was doubled such that each particle representation in the *swarm* contained 200 codons. The investigation demonstrated that this modification resulted in a significant improvement the algorithms performance.

The Chapter then introduced a variable-length form of GS, where the hard-length particle vector-length constraint is removed, thus allowing the particle representations to take on any number of codons, limited only by the memory constraints of the computer. The results of two Variable-Length GS proof of concept experiments were described. The two experiments differed only in their implementation details; the particle representation Variable-Length GS in the first was initialised to take on a maximum of 100 codon, whereas in the second experiment this was increased to 200 codons. The investigation demonstrated the feasibility of successfully generating computer programs using the Variable-Length GS.

A performance comparison based on the results of the experiments showed that by increasing the initialised particle representation size, such that all particles can potentially take on double the original number of codons, leads to an increase in performance. However, this implementation was outper-

formed by the simpler fixed-length form of GS, with particle representations taking on a vector size of `200`.

6.2 Future Work

A number of new directions for future research presented themselves throughout the course of this study. The following provides a summary of those ideas.

PRNG Investigations The PRNG study documented in Chapter 4 showed that the choice of PRNG can influence the performance of the GS algorithm on certain problems, with the stronger PRNG implementation outperforming the weaker one. The experiments were conduced using two different quality PRNGs, the (weaker) system **`rand()`** and the more stringent **`eorng`** PRNG. A possible direction for future research is to perform a more detailed study such as that conducted in [32, 32, 31, 4] where a larger number of varying quality PRNG were used to test the effects of different quality PRNGs. A more detailed study could help us make a more informed decision concerning the type of PRNG to use when implementing the GS. Such an investigation could also determine that other PRNGs could yield further performance improvements.

Fixed-Length GS This study demonstrated that increasing the number of codons in the particle repersentations to double their original size (i.e. from `100` to `200`) leads to a significant gain in performance. Therfore, increasing the number of codons even furter is a potential topic for future research. Also, a detailed analysis is warrented in the effects of this in terms of swarm behaviour and intron usage in order determine why the increase leads to such a significant performance gain.

Variable-Length GS The study presented a number of different implementation strategies. Each of these strategies had a different success rate depending on the type of problem tackled. Future work will investigate why this is so. In this initial proof of concept study we have not attempted parameter optimisation for the various variable-length strategies and this may also lead to further improvements of the Variable-Length GS algorithm.

It must also be noted that both variable-length and fixed length forms of GS are completely devoid of any reproduction operators, therefore there is potential for further enhancements to the GS by introducing concepts such as selection, crossover, replacement and mutation. Overall, the results presented are very encouraging for future development of the GS algorithm, and other potential Social or Swarm Programming variants.

Bibliography

[1] T. Back, D. B. Fogel, and T. Michaelwicz. In T. Baeck, D. B. Fogel, and Z. Michalewicz, editors, *Evolutionary Computation 1 Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, 1999.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, 1999.

[3] H. J. Bremermann. The evolution of intelligence. the nervous system as a model of its environment. Technical Report ONR report no. 1, contract Nonr 477(17), University of Washington, Seattle, USA, 1958.

[4] E. Cantú-Paz. On random numbers and the performance of genetic algorithms. In *GECCO*, pages 311–318, 2002.

[5] Cleary and O'Neill. An attribute grammar decoder for the 01 multiconstrained knapsack problem. In *European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP), LNCS*, volume 5, 2005.

[6] J. Daida, S. Ross, J. McClain, D. Ampy, and M. Holczer. Challenges with verification, repeatability, and meaningful comparisons in genetic programming. In J. R. K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 64–69, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[7] C. Darwin. *On the Origin of Species by Means of Natural Selection.* Murray, London, 1859. 6th (final) ed'n 1872.

[8] K. Entacher and S. Wegenkittl. The plab picturebook: Load tests and ultimate load tests, part II: Subsequences report, Feb. 05 1997.

[9] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, New York, 1966.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[11] J. Heinrich. Detecting a bad random number generator. Cdf6850 statistics, University of Pennsylvania, 2004.

[12] M. Hemberg and U.-M. O'Reilly. GENR8 - using grammatical evolution in A surface design tool. In A. M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 120–123, New York, July 2002. AAAI.

[13] F. Heppner and U. Grenander. *In S. Krasner, Ed., The Ubiquity of Chaos.* AAAS Publications, Washington DC, 1990.

[14] J. H. Holland. Hierarchical descriptions of universal spaces and adaptive systems. Technical Report ORA Projects 01252 and 08226, University of Michigan, Ann Arbor, 1968.

[15] J. H. Holland. *Adaptation in natural artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[16] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm Intelligence.* Morgan Kauffman, San Mateo, California, 2001.

[17] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proc. of the IEEE Int. Conf. on Neural Networks*, pages 1942–1948, Piscataway, NJ, 1995. IEEE Service Center.

[18] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

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

[20] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. 1994.

[21] W. B. Langdon. Evolving data structures with genetic programming. In *ICGA*, pages 295–302, 1995.

[22] W. B. Langdon. Fitness causes bloat in variable size representations. Technical Report CSRP-97-14, University of Birmingham, School of Computer Science, 14 May 1997. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97.

[23] W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[24] W. B. Langdon and R. Poli. Why ants are hard. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. 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.

[25] S. Luke. Code growth is not caused by introns. In D. Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 228–235, Las Vegas, Nevada, USA, 8 July 2000.

[26] G. Marsaglia. DIEHARD: A battery of tests of randomness. Technical report, Florida State University, Tallahassee, FL, USA, 1996.

[27] M. Matsumoto. Twisted GFSR generators II. *ACM Transactions on Modeling and Computing Simulation*, 4, 3:266–254, 1994.

[28] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM trans. on modeling and computer simulation*, 2, 3:179–194, 1992.

[29] M. Matsumoto and T. Nishimura. *Dynamic Creation of Pseudorandom Number Generators.* Springer-Verlag, New York, 1998.

[30] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, Jan. 1998.

[31] M. M. Maysenburg and J. A. Foster. The effect of the quality of pseudo-random number generators on the performance of a simple genetic algorithm. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[32] M. M. Meysenburg and J. A. Foster. Random generator quality and GP performance. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1121–1126, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[33] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, third edition, 1996.

[34] J. H. Moore and L. W. Hahn. Systems biology modeling in human genetics using petri nets and grammatical evolution. In *GECCO (1)*, pages 392–401, 2004.

[35] P. Naur. Revised report on the algorithmic language ALGOL 60. *Commun. Assoc. Comput. Mach.*, 6:1–17, 1963.

[36] P. Nordin and W. Banzhaf. Complexity compression and evolution. In *ICGA*, pages 310–317, 1995.

[37] M. O'Neill. *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. PhD thesis, University Of Limerick, Ireland, aug 2001.

[38] M. O'Neill and A. Brabazon. Grammatical swarm. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 163–174, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[39] M. O'Neill and A. Brabazon. *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2005.

[40] M. O'Neill, A. Brabazon, M. Nicolau, S. McGarraghy, and P. Keenan. πgrammatical evolution. In *GECCO (2)*, pages 617–629, 2004.

[41] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE-EC*, 5:349–358, Aug. 2001.

[42] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[43] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in grammatical evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, mar 2003.

[44] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, Oct. 1988.

[45] W. H. Press. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, USA, second edition, 1992.

[46] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzbog, Stuttgart, 1973.

[47] J. Riget and J. S. Vesterstrøm. A diversity-guided particle swarm optimizer – the ARPSO. Technical Report 2002-02, EVALife, Dept. of Computer Science, University of Aarhus, 2002.

[48] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *First European Workshop on Genetic Programming 1998*, pages 83–95, Berlin, 1998. Springer.

[49] Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *IEEE International Conference on Evolutionary Computation*, Anchorage, Alaska, USA, 1998.

[50] Y. Shi and R. C. Eberhart. Parameter selection in particle swarm optimization. In *Evolutionary Programming*, pages 591–600, San Diego, USA, 1998.

[51] T. Soule. Exons and code growth in genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 142–151, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[52] T. Soule and J. A. Foster. Effects of code growth and parsimony presure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, 1999.

[53] T. Soule and R. B. Heckendorn. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 3(3):283–309, 2002.

[54] J. von Neumann. Various techniques for use in connection with random digits. In *von Neumann's Collected Works*, volume 5, pages 768–770. Pergamon, 1963.

[55] M. Woodger. An introduction to ALGOL 60. *The Computer Journal*, 3:67–75, July 1960.

Index

C_{Max}, 41, 66, 78
C_{Min}, 41, 78
g_{best}, 35, 84, 85
$itermax$, 44
p_{best}, 35
`AND, NOT, OR`, 52
`PID`, 66
`RAND_MAX`, 63
`eorng.h`, 59, 66
`eorng`, 67, 69, 73
`non-terminals`, 77
`rand()`, 6, 62, 65, 67, 69, 73
`random()`, 62
`randu`, 62
`srand()`, 63
`terminals`, 77
3 Multiplexer, 52, 73

adaptive search, 10
Adenine (A), 21
amino acids, 22
Avram Noam Chomsky, 23

Backus Naur Form (BNF), 19, 24
benchmark, 4
bias, 60
binary string, 21
binary values, 17
biological organism, 18
Biological System (BS), 20
bit shifting, 66
bit-flipping, 18
bloat, 3, 76, 99, 109
BNF, 23
BNF grammar, 26, 31, 52
bounded, 5

C/C++, 62, 65
candidate solution, 46
cellular units, 21
chromosomes, 15, 18
classification, 10
closure, 17
code breaking, 53
codebreaker, 53
coin flip, `flip()`, 66
combinatorial optimization, 10
Context Free Grammars, 24
Context Sensitive Grammars, 24
contributions, 4

control parameters
- self confidence,$c2$, 38
- sociality,$c1$, 38

converge, 3
crossover, 17, 39
- bit-string, 18
- single point, 14
- two point, 14
- uniform, 14

Cytosine (C), 21

Darwanian Evolution, 10
degrade, 4
Deoxyribonucleic acid (DNA), 21
design, 10
deterministic, 61
diehard tests, 65
diversity, 14
DNA strands, 21

environment, 10
equidistribution, 64
Erick Cantu-Paz, 68
evolution, 7, 9
Evolutionary Algorithms (EAs), 10
Evolutionary Computation (EC), 9
Evolutionary Programming (EP), 10
Evolutionary Strategies, 15
Evolutionary Strategies (ES), 10
execution penalty, 49
exploration, 14

finite state machines, 11
fitness (F), 13
fitness function, 46
fitness value, 12, 69
fixed-length, 2–4, 8, 17, 75, 76
floating point, 42
fluff, 99
formal grammars, 23
Frank H. Heppner, 33
future development, 116

Gaussian mutation, 15
GE - Mapping Process, 46
gene, 18
generation, 10, 12, 13
genes, 21
Genetic Algorithm (GA), 10, 60
Genetic Programming (GP), 10
genome, 12, 28, 76
genotype, 26
genotype-phenotype, 39
genotypic, 2
Grammatical Evolution (GE), 5, 9, 18
Grammatical Swarm (GS), 39
Guanine (G), 21

hardware, 60
hybrid algorithm, 39

individual, 12
individual traits, 10
inertia weight, ω, 37
Ingo Rechenberg, 10, 15
initialization process, 34
input-output, 51
integer, 2
intermediate recombination, 15
introns, 77

James Kennedy, 34
Joel Heinrich, 62
John Backus, 24
John Holland, 11
John Koza, 11

landscape, 14, 60
Lawrence Fogel, 11
linear genome, 19
local optimum, 14
logical circuit, 52
lookahead function, 49

Maarten Keijzer, 65
machine learning, 10
mapping process, 77
Martin Mersenne, 65
Mastermind, 53, 72
Matsumoto, 64
Maximum Velocity, V_{Max}, 40
memory constraints, 61
Mersenne Twister, 64
Mersenne Twister (MT), 6, 59
mRNA, 22
Mutation, 14
mutation, 39, 60

natural selection, 9, 58
nature, 6
Nishimura, 64
non-terminal, 24
nucleotide, 21

offspring, 13, 15
Operators
  Θ_m, mutation, 12
  Θ_r, recombination, 12
  Θ_s, selection, 12
optimisation, 5
optimisation algorithm, 33

parent, 13
parse trees, 11
Particle Swarm Algorithm (PSO), 34
Particle Swarm Optimisation (PSO), 9, 32, 40
Peter Naur, 24
phenotype, 20, 23, 26
pins, 53

population-based, 4, 5, 99
premature convergence, 14
PRNG, 4, 6, 59, 61–64, 67, 114, 115
problem space, 34
production rules, 24
program generation, 39
protein, 22
proteins, 21
pseudo-random, 64, 66

Quartic Symbolic Regression, 50

random numbers, 60, 61
randomness, 7, 60
Recombination, 13
recombination, 14, 60
Regular Grammars, 24
representations, 5, 60, 75, 99
reproduction operators, 17
ribosome, 22
Robert C. Eberhart, 34
roulette selection, 13
rRna, 21

Santa Fe Ant Trail, 69, 74
scattering, 60
search heuristic, 2
search mechanism, 39, 44
search space, 34
seed, 61, 63
Selection, 13
selection, 60
selection operator, 13
simulation, 60, 77
social learning, 1
Social Programming, 6, 39, 116
social-psychological, 6
software, 60
species, 11
stochastic, 4, 57, 60
Strategy I, 84
Strategy II, 84
Strategy III, 85
Strategy IV, 85
string, 17
subtree crossover, 17
swarm, 77
Swarm Programming, 39, 116
symbolic regression, 50
system clock, 63
system-supplied, 62

target function, 50
terminal symbols, 24
time step, 49
tournament selection, 13
transcription, 20, 21
translation, 20
translation process, 31
tree representation, 11

tRNA, 22

Truth Table, 52

Twisted Generalised Feedback Shift
Register (TGFSR), 64

Tyrosine (T), 21

uint32 typedef, 66

UNIX, 62

Unrestricted Grammars, 24

variable-length, 2, 3, 5, 83, 99, 111

variable-sized, 16

vector, 3–5, 75

vector-length, 75

velocity, 34, 60

velocity update, $v_i(t + 1)$, 36

whole numbers, 42

wrapping operator, 43