Random Number Generation Using Genetic Programming

by

Philip Leonard
200714801

A Dissertation

Submitted to

The University of Liverpool

in partial fulfillment of the requirements for the degree of

BSc Computer Science

April 28, 2014

Contents

List of Figures

List of Tables

1 Abstract

This is the final report for the project; 'Random Number Generation Using Genetic Programming' by Philip Leonard, supervised by Dr David Jackson (primary supervisor) and Professor Paul Dunne (secondary supervisor). This document covers all aspects of the project from the introduction and background, to the design, implementation and analysis of the project in order to give the reader an understanding of how I implemented the project and what discoveries I made when evaluating the project.

This report demonstrates that using methods described by Koza in [1], it is possible to genetically breed Random Number Generators (RNG) that produce sequences of pseudo random bits with near maximal entropy, using the Genetic Programming (GP) paradigm. All of the steps for designing and implementing the GP approach are covered in detail in this report. The results I obtained for the GP approach match those produced by Koza. This allowed me to then make fair comparisons to the other methods of producing RNGs, by both evolutionary and conventional methods.

This report more importantly shows that it is possible to produce better results using the Single Node Genetic Programming (SNGP) methodology described by Jackson in [2]. Using the SNGP paradigm, it possible to obtain smaller and higher entropy RNGs, (on average) six times quicker and at twice the rate of Koza's standard GP implementation manages to achieve. The stages of designing, implementing and evaluating the SNGP approach, like the GP approach, are covered in detail in this report.

As well as this, this report also demonstrates that the Random Number Generators Produced genetically by these methods also produce sequences of random bits with higher entropy than other widely used RNGs. It not only outperforms the C programming languages Linear Congruential Generator (LCG) algorithm implemented in the *rand*() function, but can also perform better than a Hardware/"True" Random Number Generator that creates random bit sequences from observed atmospheric noise.

This report also shows how a RNG produced by these evolutionary means might be implemented for use in the C programming language, by showing a pragmatic approach for developers to achieve higher entropy random numbers for use in their applications and software. I also show however that this comes at the cost of increased computation time compared to the C *rand*() function.

At the end of the document I also discuss some further research that can be carried out in order to find ways of further improving the results I obtained by modifying the GP and SNGP implementations. I also discuss the learning process through this project and explaining the new skills that I have acquired. I go onto discuss how this project conformed to the British Computer Society: Chartered Institute for IT's code of conduct and code of practice (where it is applicable).

All of the code for the implemented software can be found in the appendix of this report on the CD provided. As well as this, all of the pdf documents produced for this project and the LaTeX files used to create them can be found there, as well as all of the test data used in evaluating all of the approaches.

Keywords: Genetic Algorithm (GA), Genetic Programming/Program (GP), Pseudo Random Number Generator (PRNG), Random Number Generator (RNG), True Random Number Generator (TRNG), Entropy, Single Node Genetic Programming/Program (SNGP), Crossover, Mutation, Fitness Proportionate Reproduction (FPR), Linear Congruential Generator (LCG).

Word Count: 14965 words[1]. Not including words in contents, appendix, bibliography, diagrams, equations, headings or code listings.

[1]Words counted using *TexCount* web service (v3.0.0.24) : http://app.uio.no/ifi/texcount/online.php

2 Introduction

Random Number Generators have many computational, scientific and societal applications. They are essential in; gambling, lottery draws, cryptography, statistical sampling, Monte Carlo simulations, randomized algorithms, password generation and other applications where an unpredictable result is desired. Randomness can be interpreted in more than one way and has no strict mathematical definition. There are a series of tests that can be run on a set of data in order to evaluate some conception of randomness. These include the gap test, frequency test, runs test and information entropy test amongst others described in the NIST[2] "*Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*" [3]. Whilst there isn't a way to prove randomness, these statistical tests are a good way of analysing randomness in the eyes of the user and for their practical application. In this project, randomness is tested using the Shannon Entropy equation [1, p.2], which is a measure of the unpredictability/uncertainty in a random variable. A Pseudo Random Number Generator (PRNG) is a program/algorithm which generates numbers that possess the characteristics of a "truly" random number. These programs typically employ seed values (an example being the number of nanoseconds since the Unix epoch[3]), or they use a sequence of consecutive numbers as an input. Mathematical operators are applied to this input in order to fabricate a "random" number or bit sequence. This differs from True Random Number Generators which typically use a piece of hardware to measure an unpredictable environmental occurrence such as atmospheric noise measured using a radio antenna [15] or the quantum effects on photons passing through a semi-transparent mirror[14].

The Genetic Programming paradigm is an evolutionary method of solving problems, where the implementer can define a task where solutions can be randomly generated and tested by some fitness function. The population can then be evolved based on their fitness in order to search through the problem space to find solutions which meet the fitness requirements of the implementer.

Evolving Random Number Generators is therefore an applicable problem instance for Genetic Programming, as there is no deterministic way of producing a RNG with high entropy. The aim of the project is to assess two Genetic Programming methodologies in order to determine which is best suited for genetically breeding RNGs. The method described by Koza in [1] was followed in order to implement the idea of evolving a RNG in the widely known traditional Genetic Programming methodology. The next step was to implement the same idea using the Single Node Genetic Programming Methodology described by Jackson in [2]. The effectiveness of both where then compared by gathering data from both implementations and conducting some cross examination to determine the victor. In order to determine the effectiveness of the RNGs as a real world solution, entropy data was gathered from pre-existing and widely used RNGs. These were namely the C rand function, which is a PRNG that makes use of a Linear Congruential Generator algorithm in order to produce random numbers, and a TRNG was also evaluated in order to compare the effectiveness of the genetically produced RNGs against a non-deterministic source of random data.

Attempting to match Koza's implementation was a one of the major challenges of the project. Koza doesn't give many details of how he implemented the Genetic Program. Instead he gives a theoretical overview of the operations, and then the results of his implementation. Therefore a good understanding of the GP paradigm template was required in order to take Koza's words and implement them successfully as a working program.

In terms of software, three programs have been produced. The GP implementation and the SNGP implementation form two separate single threaded command line programs. The command line interface displays information and run progress to the user, and all the data for each generation and each run is written to log files. The third program developed is to test both the C random number generator and the TRNG and to write the entropy data to log files as well. From these implementations I was then able to start collecting data. In order to gather sufficient data for analysis, I ran both the SNGP and GP implementations 50 times and also ran 50 tests of the C rand function and the TRNG.

The effectiveness of the solution can be viewed in two ways. To begin with, both methods are viable ways to produce RNGs by evolutionary means. Therefore the implemented software was a successful venture. The other and perhaps more important evaluation of how effective the solutions are is whether the SNGP proves a more effective way of producing RNGs than GP, and whether they stand up to other widely used RNGs (both deterministic and non-deterministic). The results of the project are very promising. To begin with, SNGP proves a much faster way of evolving RNGs, finding solutions on average over 6 times quicker than the standard GP methodology. Not only is it quicker, but it also has a solution rate twice that of the Genetic Programming Methodology. The solutions produced by SNGP are also much smaller than those produced by GP as branches can be repeated but there values are only calculated once and then stored dynamically. The solutions produced by the two genetic methodologies also outperformed the C *rand*() function and even the TRNG.

[2]National Institute of Standards and Technology

[3]00:00:00 (UTC), Thursday, 1 January 1970

3 Background

This project is based on two pieces of research into Genetic Programming from two separate research papers. The first paper by Koza entitled "Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm" is the main basis for the project. It describes how Random Number Generators can be evolved using Genetic Programming. Koza describes the design for the implementation, and displays his results and findings. Koza concludes that it is possible to genetically breed randomisers with high entropy. The aim of the first part of this project was to take this implementation design and implement it in the C programming language, in order to try and match his results and outcomes.

The second paper used as the foundation for this project is Jacksons paper entitled "A New, Node-Focused Model for Genetic Programming" [2], where he introduces a new Genetic Programming paradigm known as Single Node Genetic Programming. Instead of using a population of tree structures like that of traditional GP, SNGP uses a graph population, where every node in the graph represents the root of a tree in it's own right. In this paper, Jackson introduces the SNGP model, and evaluates it's performance over a number of test cases which he also implemented in the GP paradigm. Jackson shows that SNGP outperforms GP in terms of solution rate, speed (efficiency) and the size of the solutions produced. The aim of the second part of this project is to further prove SNGPs computational superiority over the standard GP paradigm, again using the basis of evolving a RNG introduced by Koza in [1], in order to create a SNGP variant of the same problem.

After implementing the GP variant of generating RNGs by evolutionary means, I was able to step back and compare the results I had obtained to that of Koza's. As stated previously, trying to obtain the same results as Koza in [1] was one of the main challenges of the project. I was able to achieve individual solutions with fitness/entropy that matched the fittest that Koza produced in [1, p.6] of 27.996 out of 28.00 maximal bits of entropy. In general, the number of generations required to find solutions were greater in my implementation as opposed to the results produced by Koza. All other observations made by Koza however matched the ones I obtained from my implementation. For example, Koza explains that in the run that generated his fittest solution; *"entropy reached and slowly improved within the 27.800 to 27.900 area"*. In all runs of my implementation I observed the same phenomenon, and is in fact here most runs spent a large number of generations searching. I also obtained similar sized solutions to Koza's results.

As discussed in the specification and design documents, research was the dominating factor in preparing for this project. I began by researching both Genetic Algorithms [4, p.1 - 24] and Genetic Programming [6, p.1 - 35] [7, p.73 - 191] topics. I then researched more specifically into Genetic Programming, including a C implementation of Multiplexer Problem described in [8]. The focal point of my research has been around the papers by Koza [1], and Jackson [2]. In [1], Koza conducts some tests of the genetically produced RNGs against RNGs that were widely used at the time[4]. I also felt that to show the success of the RNGs produced by GP and SNGP in my implementations, I should do the same. I therefore did some research of current widely used RNGs today. I chose the C *rand*() function as a comparative PRNG, and also a "True" Random Number Generator, specifically from the random.org website [15]. As the TRNG generates numbers by non-deterministic means, I felt it was a strong opponent to test against the (deterministic) PRNGs produced by evolutionary means. The requirements for this project can be defined by 4 main objectives;

1) Replicate the program described by Koza in [1], in the C programming language.

2) Deliver the same idea, but by using the Single Node Genetic Programming paradigm as described by Jackson in [2], again as a C program.

3) Deliver an analysis and a conclusion of the best method of genetically breeding a randomiser.

4) Produce a final program testing the entropy of the C *rand*() function and the TRNG at random.org, and deliver an analysis of the best practical method of producing random numbers.

Design and Implementation of these requirements are to be conducted in this order.

[4] Koza's paper was published in 1991

4 Design

In this section we discuss the design of the project. As the design document produced for this project provides a full overview of the design, in this section we shall take a look at the most important elements of the design, and the changes which have been made since that document was produced. For further details on the design, the full document can be found in the appendix at the end of this report.

4.1 GP Approach

This section covers the design for the GP approach of evolving a RNG.

The goal is to genetically breed a RNG to randomise a sequence of binary digits. The input to the RNG will be the sequence J, which will run from $i = 1$ to 2^{14}. The output will be a sequence of random binary digits of size J. The binary digit at location i will be determined by the value of the LSB[5] from the output of the RNG on the i^{th} input.

As explained in [6, p.19-27] to begin designing a GP, the first step is to define the set of terminals and the set of functions, of which the program trees will be comprised of. The terminal set will be;

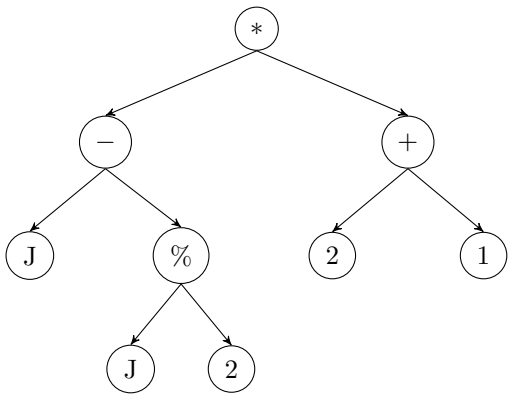
$$T = \{J,\ \Re\}$$

J as already discussed will be the value in the sequence 1 to 2^{14}, and $\Re$ will represent a small random integer constant from the set $\{0, 1, 2, 3\}$. The set F is our function or operator set;

$$F = \{+, -, *, /, \%\}$$

In this set, / is the protected integer quotient function and % is the protected modulus function, where both use the protected division function that returns 1 when attempting to divide by zero, and otherwise returns the normal quotient. All the terminals have an arity of 0, and all the functions have an arity of 2.

Figure 1: RNG binary tree



The next step is to define the structure of the programs. Members of the population will be represented as binary trees, in the form of prefix expressions. Take for example the expression;

$$*\ -\ J\ \%\ J\ 2\ +\ 2\ 1$$

This prefix expression can be represented as the binary tree in figure 1. And vice versa, the prefix expression above can be generated from a pre-order traversal of the binary tree in figure 1. For experimentation, the user will initially define the major parameters of the GP run. These parameters are defined in table 1 below;

Table 1: Input parameters

Data Type	Input description
Integer	Population size
Integer	Maximum number of generations
Integer	Maximum initial program depth and maximum crossover program depth
Double	Fitness proportionate reproduction to crossover operation balance
Double	Internal to external node crossover probabilities
Double	Target fitness value

Now we have defined the parameters, the terminals T, the functions F, and the structural representation of the programs, the next step for the GP is to use these definitions to enumerate an initial random population of programs.

Defined by the user, the GP will generate a population of the given size, where each program in the population has a maximum initial tree depth of the given size. In order to generate a wide and genetically diverse initial population, in [6, p.11-14] Koza suggests the even combination of both "grow" and "full" methods to generate program trees known as the "ramped half and half method". This allows for a range of tree shapes and sizes all the way from a single terminal node program such as J, all the way up to a program which would be produced by the full function. Using this ramped half and half method, we can obtain half a population of large programs, and another half of programs with varying shapes and sizes.

[5] Least Significant Bit. The lowest (furthest right) bit in a binary sequence determining whether the number is odd or even.

Now that the GP has obtained an initial random population, it will evaluate each program and calculate it's corresponding fitness. As discussed above, the output of the RNG is a sequence of random binary digits of length J. To calculate the output, we must run through the sequence J from $I = 1$ to 2^{14} substituting the occurrences of J in the program with i. The LSB of the output given $J = i$ corresponds to the value in the binary sequence at position i. The GP will then calculate the binary output sequence for each RNG in the population.

Now that the GP has evaluated every RNG, it must use a fitness function to calculate the fitness for each RNG given its respective output. In order to measure how "random" the output is from a RNG, their are a variety of statistical and mathematical techniques at our disposal as discussed earlier. In this case, we are going to use the Shannon entropy equation from the field of Information Theory as our fitness function. The entropy of a sequence of binary digits is the equality in the occurrence of a set of binary sub-sequences of length h in that sequence. The Shannon entropy calculation for all possible 2^h binary sub-sequences of length h is;

$$E_h = -\sum_j P_{(hj)} \ \log_2 \ P_{(hj)}$$

Here j ranges over all possible sub-sequences of length h in the binary sequence (the RNG output in this case). In order to achieve maximum entropy for E_h, all probabilities for all 2^h binary sub sequences of length h must be equal to $\frac{1}{2^h}$.

To calculate the entropy for the output of every RNG, we want to evaluate it for more than one binary sub sequence length. This is achieved by calculating the summation of the formula above for E_h but over a range of lengths of h from 1 to N_{max};

$$E_{total} = \sum_{h=1}^{N_{max}} \left[-\sum_j P_{(hj)} \ \log_2 \ P_{(hj)} \right]$$

The maximum possible entropy value for sub sequences from 1 to N_{max} can be calculated using the following sum;

$$\begin{aligned} E_{max} &= \sum_{i=1}^{N_{max}} i \\ &= \frac{N_{max}(N_{max}+1)}{2} \\ &= \frac{7 \cdot (7+1)}{2} \\ &= 28 \end{aligned}$$

In this GP implementation, We shall be using sub sequences of lengths 1 to 7, so $N_{max} = 7$. This will give us the maximum entropy and therefore the maximum fitness of 28.

Now that the fitness function has been defined. The entropy for every RNG's output is calculated, for sub sequences of lengths 1 to 7. In the C implementation, each program (RNG) will be stored in an individual struct. Each struct will contain; the RNG code/prefix expression, the program length, the output binary sequence, the scalar entropy break down for each of the sub sequence lengths of h from 1 to $N_{max} = 7$ and the total fitness (the sum of these scalar entropies). So far in the design, all of these pieces of information have been defined. The population of all these structs are contained in another "population struct" of the size defined by the user. Representing the programs and the population like this allows for easy manipulation and portability between methods in the GP.

Figure 2 is a simple flow chart representation for the GP. On every round of the algorithm, data is gathered for that particular run and is saved to a text file in the format of a CSV[6](Comma Separated Values) file. Each output and it's type are defined in table 2 below;

[6]Where values are delimited using commas i.e. $w, \ x, \ y, \ z, ...$

Table 2: Output data

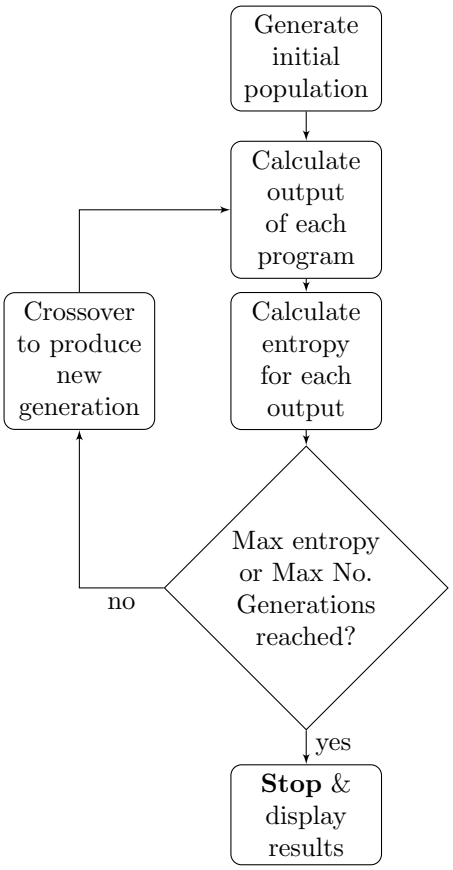
Data Type	Output description
Integer	Generation number
Double	Generation run time in seconds & hundredths of a second
Double	Average entropy/fitness for the population
Double	Total entropy/fitness of fittest candidate
Double	Entropy for subsequence size 1 of fittest candidate
⋮	⋮
Double	Entropy for subsequence size 7 of fittest candidate
Boolean	Entropy of fittest candidate is $\geq$ target fitness?
String	Prefix expression for fittest candidate

Since the design document, I have added in one extra piece of output data. This is the "Average entropy/fitness for the population", and I included it so that during the analysis phase, I could determine how well the fitness of the population as a whole was growing and not just a single candidate. This comma separated data will be appended to the text file for each generation of the GP (each preceded with a return line character). For further information about the design of the structure of the output data, please refer to the design document in the appendix.

The final process in the GP to define is the genetic operation which shall be used. Two main genetic operators used in GP are mutation and crossover [6, p.15-17]. Both operate by altering program trees in different ways in order to create offspring programs. In this GP, I will be using the subtree crossover operation alone, and I will not be implementing subtree mutation in order to conserve computing time[7].

Subtree crossover works by taking two parent program trees and joining them to create two offspring programs. This is done by selecting a crossover point (node) in each parent tree. Both of the subtrees at the root of the crossover point are then swapped to generate the two new offspring. Crossover point selection can be at any node in the tree including leaf (in this case terminal) nodes. For this GP, one of the user inputs to the program in figure 1 on page 6 is the probability divide of selection of internal to external nodes. In [7, p.114], Koza describes the benefits of using a 90% internal (function) and 10% external (terminal) split, saying that it "promotes the recombining of larger structures" so that simple swapping of terminals like that of point mutation is less probable.

Figure 2: GP flowchart



In table 1 on page 6, there is a parameter "Fitness proportionate reproduction to crossover operation balance". This input determines the split between Fitness proportionate reproduction (FPR) and crossover. FPR is where both parents are selected for crossover relative to their fitness. In the regular crossover selection, one of the parents is selected relative to their fitness and the other is selected with an equal probability amongst the rest of the population. The following equation calculates the selection probability for the i^{th} individual in the population (used for selecting both parents in FPR and one in crossover);

$$p_i = \left(\frac{f_i}{\sum_{x=1}^{|P|} f_x} \right) \qquad \text{where;}$$

p_i - selection probability of i
f_i - fitness value of i
f_x - fitness value of x
$|P|$ - population size

Now that the GP has generated a new *evolved* population, it returns back to step 2 in figure 2 on page 8. This process continues until one or both of the termination criteria are met as discussed above.

[7] Subtree mutation requires the generation of new subtrees which can prove to be computationally intensive, especially when mutation probability is high

Above is a literal design of the GP, covering all the major aspects of the operations. Using the definitions above, a pseudocode can be created for the key methods in figure 2 on page 8 which bears a closer functional resemblance to the actual C implementation of the GP. All algorithms produced in the design process can be found in the design document. Here I only introduce the changes made and the new algorithms introduced after the design document was written.

In algorithm 3 we can see that a minor change has been made since the design document was written. Instead of using a brute force pattern matching algorithm, I have for the efficient finite automaton based KMP pattern matching algorithm (Named after it's designers, Knuth, Morris and Pratt). Algorithms 1 and 2 describe the two functions of this algorithm. The Compute-Prefix-Function creates a finite automaton transition function for the pattern that it is given. In essence this function calculates the earliest place in the pattern where the pattern can reoccur in itself. This way time is saved matching what has already been matched. The matching algorithm itself feeds the pattern through this automaton, keeping track of its state.

Algorithm 1 Compute-Prefix-Function(j_2)

Input: Binary pattern j_2.
Output: The finite automaton mapping function π.

```
1: m ←length(j_2)
2: π[1] ← 0
3: k ← 0
4: for q ← 2 to m do
5:     while k > 0 ∧ P[k + 1] ≠ P[q] do
6:         k ← π[k]
7:     end while
8:     if P[k + 1] == P[q] then
9:         k ← k + 1
10:    end if
11:    π[q] ← k
12: end for
13: return π        ▷ This algorithm returns the function π where π[q] =max{k : P_k is a proper suffix of P_q}
```

Algorithm 2 KMP-Matching(j_2, O)

Input: Binary pattern j_2 and the text (in this case binary sequence) O.
Output: The number of occurrences of j_2 in O.

```
1: n ←length(O)
2: m ←length(j_2)
3: π ←Compute-Prefix-Function(j_2)
4: q ← 0
5: count ← 0
6: for i ← 1 to n do
7:     while q > 0 ∧ P[q + 1] ≠ T[i] do
8:         q ← π[q] ▷ Using the prefix function to find the closest move backwards in the pattern we can go
9:     end while
10:    if P[q + 1] == T[i] then
11:        q ← q + 1
12:    end if
13:    if q == m then                ▷ Reached the end of the pattern and therefore have a match
14:        count+ = 1
15:        q ← π[q]
16:    end if
17: end for
18: return count
```

Algorithm 3 Fitness-Function(O)

Input: Tree binary sequence output O

Output: Fitness value E_{total}, and scalar entropies $E_1, E_2, ..., E_7$ corresponding to the output O (which in turn corresponds to a tree/RNG)

$E_{total} \leftarrow 0$
for $h \leftarrow 1$ to 7 **do** ▷ this algorithm represents $E_{total} = \sum_{h=1}^{7} \left[-\sum_j P_{(hj)} \ \log_2 \ P_{(hj)} \right]$
    $F \leftarrow \emptyset$
    $totalOcc \leftarrow 0$

    $j \leftarrow 2^{h-1}$
    if $h == 1$ **then** ▷ we want to include 0 as a binary sequence of length 1
        $j \leftarrow 0$
    end if

    for j to $2^h - 1$ **do** ▷ this is for all numbers of binary sequence length h
        $occurrence \leftarrow$KMP-Matching(j_2, O) ▷ Brute force algorithm replaced with KMP algorithm
        $F \cup \{(j_2, occurrence)\}$
        $totalOcc \leftarrow totalOcc + occurrence$
    end for
    $E_h \leftarrow 0$ ▷ calculating entropy value for subsequence size h
    for $\forall x \in F$ **do**
        $E_h \leftarrow E_h + \left(-1 * \left(\frac{x.occurrence}{totalOcc} \log_2 \frac{x.occurrence}{totalOcc}\right)\right)$
    end for
    $E_{total} \leftarrow E_{total} + E_h$
end for
return $E_{total}, E_1, E_2, ..., E_7$

4.2 SNGP Approach

The following section of this document is concerned about the design of the SNGP implementation of evolving a RNG by means of Genetic Programming. To do this, I shall be using both [1] by Koza and [10] by Jackson. This is a summary of the design, covering its most important aspects. A more elaborative description of the proposed system and examples to go along with functions can be found in the original design document in the appendix.

Single Node Genetic Programming is similar to regular Genetic Programming in many ways, but differs in a few crucial aspects, giving rise to considerable efficiency and solution rate boosts over GP. For this reason, the design of the SNGP methodology need not be as extensive as that in section 2.1.1, but rather more to the point in terms of defining the approaches differences in comparison to GP. In [10, p.50-51] Jackson describes the SNGP model in the following way;

A population is a set M of N members where;

$$M = \{m_0, \ m_1, \ ..., \ m_{N-1}\}$$

Each member m_i is a tuple;

$$m_i = < u_i, \ r_i, \ S_i, \ P_i, \ O_i >$$

$u_i \in \{T \cup F\}$ - node in the terminal or function set
r_i - fitness of the node
S_i - set of successors of the node
P_i - set of predecessors of the node
O_i - output vector for this node

This tuple will be adapted into a *struct* in C like the GP program described above.

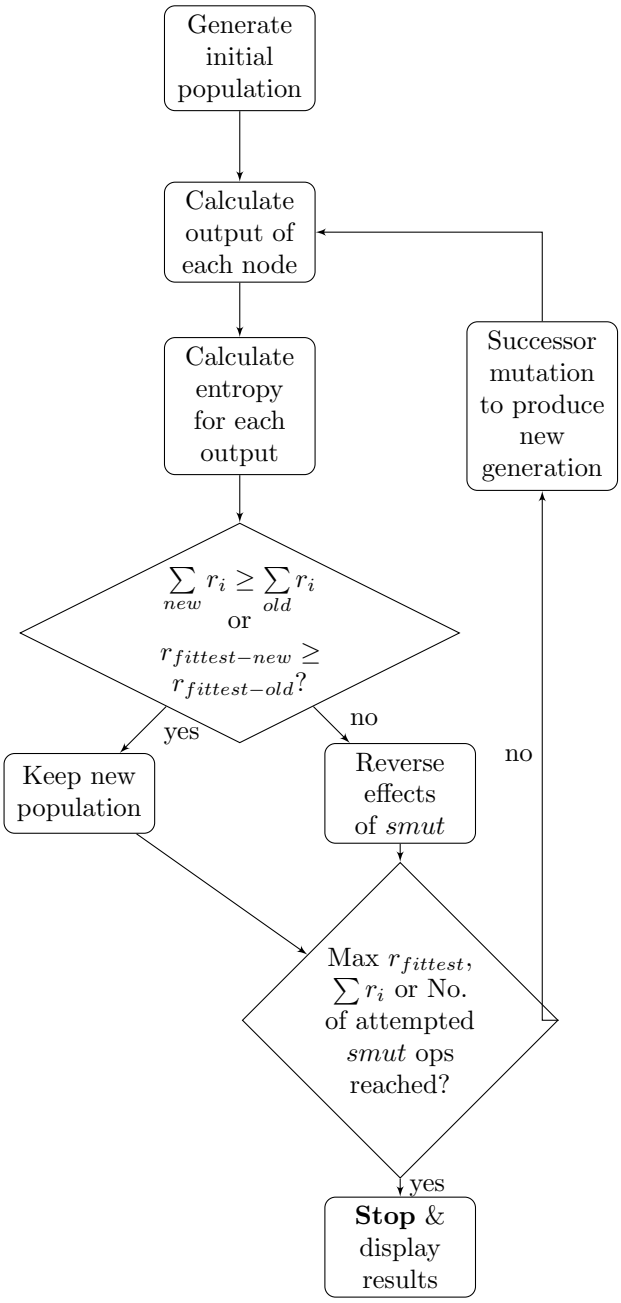
For this implementation, I will be adapting the fitness value r_i into a vector R_i [8]. The first element in the vector will be the total fitness E_{total} and the rest of the elements will be the scalar entropy values from E_1 to E_7.

SNGP is node focused, this means that members of the population are not trees, but are single nodes which together create a larger graph structure.

[8] So that the output of the SNGP will be the same as the GP, giving me the option to compare approaches on scalar entropies

Figure 3 shows a flowchart for the anticipated SNGP implementation. Here we can see that the main functions are similar to those of GP. In SNGP, the initial population is also randomly generated but instead, members of the population are chosen as individual nodes.

Figure 3: SNGP flowchart



The Inputs to the SNGP implementation are shown in table 3 on page 12. There are less inputs for the SNGP implementation compared to the conventional GP implementation. The length of a run is the equivalent to the maximum number of generations for GP, where instead the program terminates after a certain number of attempted successor mutation operations. This has been changed since the original design document. Initially termination was designed to end on the number of successful *smut* operations. It has since been changed to the number of attempted operations in order to bring fairness between GP and SNGP implementations. The terminating fitness value is taken from the sum of all fitness values in all the nodes of the population, or upon finding a solution (a root node representing a tree with an entropy greater than 27.990 bits). This has also been changed since the original design document, upon realisation that a single solution is desired. The population size is the number of nodes in the SNGP graph population. This size remains constant from initial generation throughout the rest of the run.

The SNGP population differs from the tree structure of a single member in the GP population. While the arity of functions and terminals remain the same as of that in the GP implementation, each function and terminal may have more than one predecessor. Therefore every node in the population can act as a root node for a RNG, and a population therefore contains as many RNGs as there are nodes. The reason for this coming about is due to the way that the population is initialised and evolved.

To begin with, all of the terminals in T are added to the population exactly once. From here on in over all the generations, these terminals remain the same. What changes are their predecessors. Once the terminals are added, the functions are now selected randomly and are added to the population where their operands are existing members of the population. During initialisation of the population, the output and the fitness of each individual is also assessed. As Jackson describes in [10, p.52], this is what gives rise to SNGPs efficiency.

As the first terminals are added, they are evaluated for their outputs and their fitness is subsequently calculated. In this implementation of SNGP, this process is almost identical to that of the GP implementation described above. The terminal is evaluated for all values of J from $i = 1$ to 2^{14}, and each output is stored in the vector O_i for that node. From this output vector a binary sequence of length J is generated from the LSB of each element in O_i. The fitness of that node is the total entropy for subsequence's of length 1 to 7 in the binary sequence, using the same equation as in the GP approach;

$$E_{total} = \sum_{h=1}^{7} \left[-\sum_j P_{(hj)} \ \log_2 \ P_{(hj)} \right]$$

This fitness/entropy value is stored in r_i for that node. The efficiency of SNGP arises from using the following form of dynamic programming; when functions are now added to the population and their successors are assigned, the output and fitness for these nodes are calculated from the output vectors of their successors in the set S_i, as opposed to calculating them from scratch.

In SNGP, evolution is driven by hill-climbing. This is that the fitness is taken as the sum across the whole population; $\sum r_i$. After evolution this sum is calculated again, and if it is greater (higher entropy) than the last, then this evolved population replaces the previous population, otherwise the old population is kept and the process is repeated. I have changed the original design so that it is also driven on an increasing fittest member of the population. This way, it is easier to focus the evolution on trying to produce a solution as

well as an entire graph with a high fitness.

Table 3: Input parameters

Data Type	Input description
Integer	Length of a run (number of attempted *smut* operations)
Integer	Population size (number of nodes)
Double	Total (i.e. sum of) target fitness value

Evolution of a SNGP is not done by using the crossover operation like in the GP above, but instead uses the *smut* or successor mutation operation. A member of the population (a node) is randomly selected, and one of the members in it's successor set is changed to another member of the population (still at a greater depth in the tree).

During *smut*, not all of the program nodes are effected, and therefore not all of them need to be re-evaluated for their outputs and fitnesses. As Jackson describes in [2, p.53] SNGP that retains it's efficient nature by creating an update list, where only nodes effected by smut are added. Nodes in the list are re-evaluated starting at the lowest node in the list working up to the highest node so that no node is revisited.

The outputs for the SNGP implementation are almost identical to those of the GP. Instead of Generation Number, the output here will be the *smut* operation count. Like the GP implementation I have also added the average fitness of the members of the population as an output. Please see the design document in the appendix for further detail on output data for the SNGP implementation.

Since the design document was produced, no changes have been made to the algorithms for the SNGP apart from the fitness function using the KMP matching algorithm, which is shown in algorithms 3, 1 and 2 in the design documents on pages 9 and 10. Please see the design document in the appendix for the algorithms specific to the SNGP implementation.

5 Implementation

In this section I shall cover the transition I made from design into executable code. I shall cover the GP and SNGP implementations, as well as the program I produced in order to assess the C RNG and the TRNG at random.org. As I created all of the main methods / algorithms in the design stage, the main body of implementation was to take the pseudo code and change it into C syntax. Code listings can be found in the appendix of this report, but the important functions of all approaches (algorithms in the design document implemented into code) will be described in code in this section.

5.1 GP Implementation

5.1.1 Main Data Structures

The first thing that I did when I started the implementation stage was to define the primitive objects to be used and manipulated by the algorithms described in the design section. Specifically these were the set of functions and terminals that make up the tree structures, and also the structure which holds all the information about each member of the population.

I combined the set $T = \{J, \Re\}$ of terminals and the set $F = \{+, -, *, /, \%\}$ of functions into a struct of tokens, each token was also a struct containing two bits of information;

```
struct token_entry {
   char *name;
   int arity;
};
```

The name denotes the character representation of the node, and the arity represents the number of children nodes or operands that each node has. If the node is a terminal then it will have an arity of -1, if it is a function it will have two children and therefore an arity of 1 (including 0 in the arity).

All the tokens needed to represent the RNG trees as prefix expressions are then encapsulated into an array of tokens;

```
struct token_entry token_table [] = {{"#", -1}, {"J", -1}, {"R", -1}, {"+", 1}, {"-", 1}, {"
    *", 1}, {"%", 1}, {"/", 1}, {"0", -1}, {"1", -1}, {"2", -1}, {"3", -1} };
```

The next step was to describe each member in the population as some data structure. Every member has it's own specific information, as discussed in the design section (and more extensively in the design document in the appendix).

```
struct population_member {
   char code[MAX_PROG_SIZE + 1];
   char output[J + 1];
   int proglen;
   double scalar_entropy[H_MAX];
   double total_fitness;
};
```

Every member in the population has it's own code (RNG tree represented as a prefix expression), binary output, length of the code, scalar entropy of the output and total fitness. All of this is therefore encapsulated in the struct as seen above. A population of these structs is then formed by creating a struct of a defined population size which embodies them all;

```
struct population_member population[POP_SIZE];
```

5.1.2 Generating Initial Population

In order to obtain a full population that can be tested and evolved we must randomly generate an initial population of RNG trees. These trees as discussed are to be represented in a prefix expression so that they can be easily stored in a character array but interpreted and manipulated as trees. The following "generate_initial_pop" method is called at the beginning of the programs execution. As discussed earlier and in [6, p.11-14], Koza suggests the use of the "*ramped half and half*" method. This meaning that half of the trees are created as full trees (right up to the depth limit) and the other half as grow trees (terminals are added in randomly and not just at the depth limit). I implemented my initial population function in the same way, and also with an increasing depth limit from a depth of 2 up to the user defined maximum initial depth. This gives us a diverse initial population of trees of varying shapes and sizes.

The following "generate_initial_pop" method controls the parameters for the tree types. The "full" variable is equal to the population member number modulus 2. This means that it alternates between 0 and 1, and

therefore indicating to the "create_tree" function whether the tree should be made using the grow or full method. The depth integer controls the depth limit of the trees created. This process is partitioned into groups of different tree depths;

```
void generate_initial_pop() {
  struct population_member *progp = population;
  int i, j, depth, full;
  char *end_tree;

  char* text = "Generating init pop: ";

  for (i = 2; i <= MAX_INIT_DEPTH; i++) {
     for (j = 0; j < PARTITION_SIZE; j++)  {
        full = j%2; //Half full and half grow
        depth = i;
        end_tree = create_tree(progp->code, depth, full);
        //Create tree up to ``depth" with alternating full and grow.
        end_tree = '\0';
        progp->proglen = strlen(progp->code);
        if (depth < MAX_INIT_DEPTH)
            depth++;
        progp++;
     }
   print_progress(text, i, 50, MAX_INIT_DEPTH);
  }
}
```

In this method, the population is enumerated with trees by calling the "create_tree" method. The type and depth of the trees are controlled in this method, but the tree/RNG code is created in the "create_tree" method. The implementation of this method can be seen below. If the depth is 1 then then a terminal must be added, and if the value is R then this is replaced with a random integer constant from the set {0, 1, 2, 3} and is then added to the tree code. Otherwise, as discussed above, depending on the value of the full integer, the token is chosen from the set of functions (if it is full) or from the set of both functions and terminals (if it is grow). Branches are then recursively created depending on the arity of the chosen node value. This is done with decreasing depth, as to eventually reach the depth of 1 where terminals are added instead of functions.

```
char *create_tree(char *tree, int depth, int full) {
   char token; int arity_m1, i;
   if (depth == 1) {
      token = random_terminal();
    if (strcmp(token_table[token].name, "R") == 0) {
      token = random_r(); //If the token is R then assign a random variable, either 0, 1, 2
    or 3
    }
    *tree++ = token; //Add the token to the code
   }
   else {
    token = (full ? random_function() : random_token());
    //If full is equal to 0 then function is chosen, otherwise any token from F and T can be
     chosen
    if (strcmp(token_table[token].name, "R") == 0) {
      token = random_r();
      //If the token is R then assign a random variable, either 0, 1, 2 or 3
    }
    *tree++ = token;
      arity_m1 = token_table[token].arity;
      for (i=0; i <= arity_m1; i++) //If token is a function then extra branches are created
         tree = create_tree(tree, depth-1, full);
   }
   return(tree);
}
```

From these functions, I was able to obtain an inital population. I used print statements during development of these functions in order to see that the trees being created were in fact correct and diverse.

5.1.3 Evaluating Trees

After implementing the functions that generate a random initial population, the next phase (referring to the flow chart in figure 2 on page 8) was to write a method to evaluate/calculate the output of these trees. Taking the algorithm from the design section I was able to implement the following method for recursively conducting a pre-order traversal of the RNG expressions, applying the operations present in the functions to the terminals that are returned at the leaf nodes of the tree;

```
long calculate_tree_output() {
  char code_token = *prog_code++;
  char token = *token_table[code_token].name;
  switch(token) {
    case '+': return(calculate_tree_output() + calculate_tree_output());
    case '-': return(calculate_tree_output() - calculate_tree_output());
    case '*': return(calculate_tree_output() * calculate_tree_output());
    case '%': return(protected_mod(calculate_tree_output(), calculate_tree_output()));
    case '/': return(protected_div(calculate_tree_output(), calculate_tree_output()));
    case 'J': return(current_j);
    default: return(token-'0'); //Must be R = {0, 1, 2, 3}
  }
}
```

Previously my protected mod (and division) functions were returning the following statement (if *dom* is not equal to 0);

```
return (num%dom);
```

During implementation, on some runs I was observing an Integer overflow error. I changed the types from int to the long data type, and noticed I was still getting the same error on occasion but still for Integers. Upon some research I found that if both values (despite being longs) are not casted as them, then C will conduct Integer arithmetic operations on them regardless. I then changed the return statements to;

```
return ((long)num%(long)dom);
```

and haven't since had the error message.

The method which calls this method does so by cycling through the sequence J from 1 to 16834. The "current_j" variable above is a global integer variable which is set by this higher up method. The output from the "calculate_tree_output" method populates that individuals binary output array by using the modulus function seen below;

```
long output_for_j = calculate_tree_output();
int lsb = output_for_j % 2;
if (lsb == 0)
   progp->output[x] = '0';
else
   progp->output[x] = '1';
```

This process is repeated for every integer in the sequence of 1 to 16834 and for every member of the population. After this has been done, every member of the population will have it's own corresponding output sequence, ready for it's entropy/fitness to be calculated.

5.1.4 Fitness Function

In this section I show the transition I made from the pseudo code definition of the fitness function in the design section, into a method that is used to assess the entropy and therefore fitness of each member of the population. To recap, the equation below is used in order to calculate the entropy of a binary sequence for subsequences of length 1 to N_{max};

$$E_{total} = \sum_{h=1}^{N_{max}} \left[-\sum_{j} P_{(hj)} \ \log_2 \ P_{(hj)} \right]$$

Therefore to implement this in code the method should first cycle through all subsequences of length 1 to N_{max} and count their occurrences in the binary sequence. This is done using the KMP[9] pattern matching algorithm. As discussed in the revised design, I replaced the initial brute force pattern matching algorithm in order to make the fitness function more efficient. The algorithm is called like so, where "j2" is the padded[10] binary subsequence to be searched for (i.e. the pattern), and "bit_seq" is the binary output of the current program;

```
occurrence = KMP_matching(j2, bit_seq);
```

The occurrences are then entered into an array for that specific sequence, and the total occurrence is incremented by the same amount;

[9] Knuth Morris Pratt - finite automaton based pattern matching
[10] Including preceding zeros

```
occurrence_array[array_count] = occurrence;
array_count++;
total_occ += occurrence;
```

Once this has been done for every binary subsequence of lengths 1 to H_MAX (In our case defined as 7), then we can calculate the scalar entropy for each specific subsequence length. In the part of the fitness function below, the probabilities of occurrence of each subsequence are calculated by dividing the occurrence of that subsequence by the total occurrence for that length. The inner section of the equation above is then calculated and the scalar entropy for that length is then incremented by the entropy for that specific binary subsequence. This process is repeated for every element of "ocurrence_array" (i.e. every subsequence of that length) and is then repeated for every length to obtain all the scalar entropies;

```
for (int z = 0; z < array_count; z++) {
   int occ = occurrence_array[z];
   double prob = 0;

   if (total_occ != 0)
      prob = occ / (double)total_occ;

   double e_h_increment;
   //Avoid log(0) and negative zero
   if (prob == 0 || prob == 1)
      e_h_increment = 0;
   else
      e_h_increment = (-1*((prob*log10(prob))/log10(2.0)));
   e_scalar[h - 1] = e_scalar[h - 1] + e_h_increment;
}
```

The scalar entropies are then copied over to the scalar array of that program and then summed to obtain the full entropy/fitness value. This process of assessing the fitness is repeated for every program in the population.

When I first implemented the fitness function, I noticed I was not achieving the desired fitness levels. I initially thought that it was down to a bad population. I then used the prefix expression that Koza provides as his fittest solution in [1] as a test case, with a known fitness of 27.996 bits. I noticed that was only achieving around 15 bits of entropy. From this I knew that it was down to either the evaluation method or some component of the fitness function. I then realised that I was overlooking binary subsequence patterns with preceding zeros, (i.e. For length 2 for example I was checking the occurrences of 0, 1, 10, and 11 when in fact I should have been checking the occurrences of 00, 01, 10 and 11). Once I noticed this, I implemented a method that added the correct number of preceding zeros to each subsequence. I then tested Koza's solution again and got exactly the same entropy as he did. I knew then that my fitness function was correct.

5.1.5 Evolution

For the GP implementation the only evolutionary operator I used is genetic crossover. As Koza discussed in [1], and what I described in the design section, for selection, we use FPR (Fitness Proportionate Reproduction) and then the remaining population uses equal selection with one parent selected based on their fitness. The balance between the two is defined by the user (as a definition in the code).

For FPR, the fitnesses of the population are summed and then a random number is chosen in the sums range;

```
//Adding POP_SIZE so that programs with 0 entropy still have chance
double total_select = fitness_sum + POP_SIZE;
double selection = ( (double)rand() * total_select ) / (double)RAND_MAX;
```

The population fitnesses are then summed until the random number falls in between the current nodes fitness. Thus giving fitter members a greater chance of being selected. Below, the ineteger "s" is the index in the population array of the member chosen.

```
int fpr_selection() {
  double total_select = fitness_sum; //Adding POP_SIZE so that programs with 0 entropy still
     have chance
  double selection = ( (double)rand() * total_select ) / (double)RAND_MAX;
  for (s = 0; s < POP_SIZE; s++) {
     struct population_member *progp = &population[s];
     acc_fitness = acc_fitness + 1 + progp->total_fitness;
     if (acc_fitness >= selection && selection >= old_acc_fitness) {
        break;
    }
}
```

When calling this method, for the split of our FPR seclection is conducted like so;

```
do{
    parent_a = fpr_selection();
    parent_b = fpr_selection();
} while(parent_a == parent_b);
```

The rest of the selection is then done, as mentioned above, with one parent selected based on their fitness and the other with equal probability across the population;

```
do {
    parent_a = fpr_selection();
    parent_b = rand()%POP_SIZE;
} while(parent_a == parent_b);
```

The crossover points in both parents are then selected. To begin with, the variable node_type, a random rational natural number greater than 0 and less than 1 is created. If it is less than the user defined INTERNAL_NODE_PROB (which is also a rational natural number greater than 0 and less than 1) then an internal node is selected from that parent, otherwise a terminal / leaf node is selected;

```
if (node_type <= INTERN_NODE_PROB && progp->proglen != 1) {
    char token;
    do{
        crossover_point = rand()%progp->proglen;
        token = progp->code[crossover_point];
    } while(token_table[token].arity != 1);
}
else {
    char token;
    do{
        crossover_point = rand()%progp->proglen;
        token = progp->code[crossover_point];
    } while(token_table[token].arity != -1);
}
```

This method is then called for both parents in order to select crossover points.

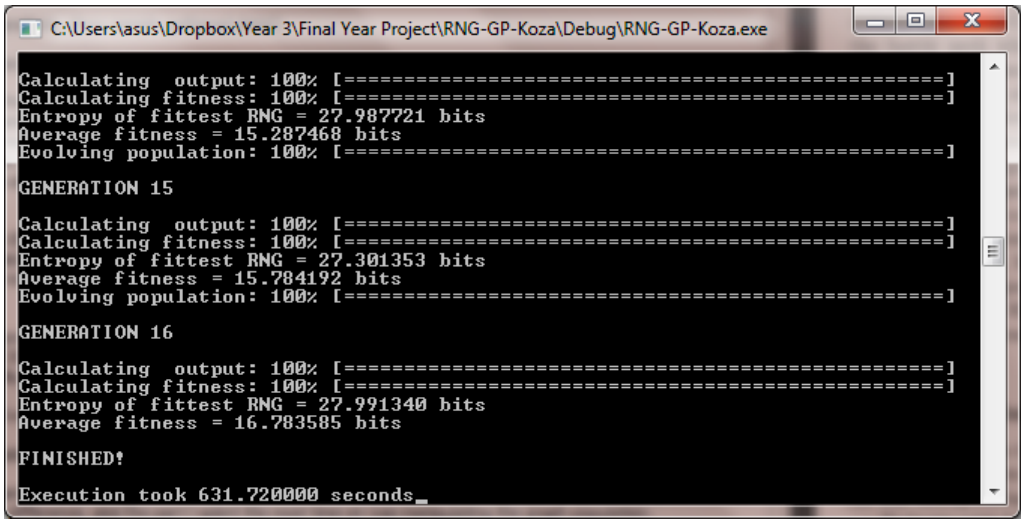
Now that I have implemented the selection of the parents and the nodes within each of those parents, the next step is to create two child programs from them. The code for creating the two children from the parents can be found in the appendix. It is a simple case of copying over the code from the two selected points into two new char arrays, but is long winded. This process is then repeated for a total of half the size of the population (each reproduction produces two child programs).

The main method of the program controls the calling and amalgamation of all the methods mentioned above. The main method repeats the process of evolving the population until the terminating criteria is met (entropy threshold reached or max number of generations reached). This is then repeated some number of times defined by the user, for conducting more runs and gathering larger amounts of data for evaluation purposes. The main method also implements a clock function which records the amount of time each run and generation takes. All of the information is written to multiple and individual run log files. Other (non major) methods and the rest of the code can be found in the appendix at the end of this document.

5.1.6 Conclusion

At the end of the GP implementation I had obtained a working command line based Genetic Program which evolves RNGs. It is adaptive to experimentation as all parameters of the run can be altered in the code, and data from each run describing each generation is written to log files on the hard drive. Having completed this, I then had a strong basis for beginning the SNGP implementation, and for collecting data about the GP performance for later evaluation. Figure 4 shows a screen shot of the GP running in the command line;

Figure 4: GP Screen Shot



5.2 SNGP Implementation

In this section I shall cover the process of implementing the SNGP paradigm of genetically breeding a RNG. As the aim of the completed program is to search for optimal RNGs in the same way as the GP implementation, a lot of the methods are the same, including the fitness function. What differs between the two implementations is that SNGP handles a graph population instead of separated trees, and evolves and evaluates them using dynamic programming.

5.2.1 Main Data Structures

In the design section and in [10, p.50-51] Jackson describes the SNGP model in the following way;

A population is a set M of N members where;

$$M = \{m_0,\ m_1,\ ...,\ m_{N-1}\}$$

Each member m_i is a tuple;

$$m_i = < u_i,\ r_i,\ S_i,\ P_i,\ O_i >$$

I implemented these tuples as structs in C. Here we can see how each node is described as a struct in C for the SNGP implementation;

```
struct population_member {
   char node_value;
   int depth;
   long dec_output[J];
   char bin_output[J + 1];
   int pred[POP_SIZE]; //1 if node index is predecessor
   int suc_1; //Array index of left successor
   int suc_2; //Array index of right successor
   double scalar_entropy[H_MAX];
   double total_fitness;
};
```

As each of these structs represents a node in a graph population, information on the nodes position is needed. Therefore I added some extra data from the GP struct. The depth of the node is important for evolution purposes, as to ensure that nodes cannot mutate to other nodes higher up or equal to the depth they are at (this could cause a cycle in the graph leading to infinite trees being created). Each node can have any number (up to POP_SIZE $- |T|$), if it's a terminal node) of predecessors, therefore the pred integer array stores the index of these predecessors. This is later used for creating update lists after evolution. Each function has two successors, the index of which are stored as integers in the struct. The decimal output vector is also needed in order to conduct dynamic calculations during evaluation.

A graph of these struct descriptions of the nodes is formed by creating a larger struct to embody them all;

```
struct population_member graph[POP_SIZE];
```

5.2.2 Generating Initial Population & Evaluation

The next stage was to write a method to populate this graph struct, called at the beginning of the programs execution. Every terminal node is added to the graph first. Terminals have no successors, and their outputs are calculated as they are added to the graph. For the J terminal, the output is 1 to 16834. For all the other terminals (i.e. $\Re = \{0, 1, 2, 3\}$) their output is an array of the terminal values (so for 0 it is an array of 16834 0's). Functions are now added to the graph, selecting successors at random. The outputs for the function nodes are also calculated as they are added to the graph, taking the outputs of the left and right hand sucessors output array at every index (through the sequence) and applying the function to these two values.

For terminals, in the code we add their values to their output array;

```
current_j = x + 1;
if (*token_table[curr_node->node_value].name == 'J') {
  output_for_j = current_j;
}
else if (*token_table[curr_node->node_value].name == '0') {
  output_for_j = 0;
}
else {
  output_for_j = *token_table[curr_node->node_value].name - '0';
}
curr_node->dec_output[x] = output_for_j;
```

This is done for every $current\_j = 1 \rightarrow 16834$. The binary array is then populated by taking the LSB of this output.

For functions, the output is calculated by taking the left and right hand nodes output and applying the operation to them;

```
struct population_member *node_left = &graph[curr_node->suc_1];
struct population_member *node_right = &graph[curr_node->suc_2];
```

The following code is the case statement for the multiplication function. The other case statements are similar in structure but they apply their corresponding operation to the two decimal values;

```
case '*':
   for (int i = 0; i < J; i++) {
       curr_node->dec_output[i] = node_left->dec_output[i] * node_right->dec_output[i];
       if ((curr_node->dec_output[i] % 2) == 0)
         curr_node->bin_output[i] = '0';
       else
         curr_node->bin_output[i] = '1';
   }
   break;
```

This process is repeated for any new node added to the graph population during it's initialisation. However, as we will see below, in later generations when the graph is updated after a mutation, not every node needs to be updated.

5.2.3 Successor Mutation / Evolution

The only evolutionary operator I used in the SNGP implementation (and described by Jackson on [2]) is successor mutation or *smut*. A node is chosen at random (minus the terminals), and one of it's successors is mutated to some other node in the graph (still at a lower depth). The implementation of this function was relatively simple as it purely relies on random selection and updating of the nodes involved;

```
int mut = (rand() % (POP_SIZE - NUM_TERMINALS)) + NUM_TERMINALS; //Randomly pick node in
    graph (excluding terminals)
int suc = rand() % 2; //Left or right hand successor

...

do {
   mut_to = rand() % mut;
   mut_to_node = &graph[mut_to];
} while (mut_to == current_suc_1 && mut_to_node->depth >= mut_node->depth);

mut_node->suc_1 = mut_to;
mut_to_node->pred[mut] = 1;
```

The nodes depth is then updated by taking the successor with the greatest depth and adding 1;

```
if (left->depth >= right->depth)
    mut_node->depth = left->depth + 1;
  else
    mut_node->depth = right->depth + 1;
```

The tricky part was to create an update list. After the successor mutation operation, the only nodes that need to be updated are the ones which are it's predecessors (parents, grandparents, etc.). Therefore I needed to create a recursive algorithm that backtracked through the graph in order to create an update list.

First I wrote the following method to initialise the update list;

```
void init_update_list() {
  for (int x = 0; x < POP_SIZE; x++) {
    update_list[x] = -1;
  }
}
```

Then I created the following method to create the update list. Given the index of the node which has been updated, the method is then called recursively on the parent nodes.

```
void create_update_list(int update_node) {
  //Only need to update the predecessors of the
  struct population_member *curr_update = &graph[update_node];

  for (int i = 0; i < POP_SIZE; i++) {
    if (curr_update->pred[i] == 1) {
      if (update_list[i] != 1) { //Adding to list (only counting repeated nodes once)
        update_list[i] = 1;
        update_size++;
      }
      create_update_list(i); //Recursively calling on predecessors
    }
  }
}
```

After implementing this I had complete code for running one generation. The rate at which the SNGP implementation evolved was alarming at first having been used to the slower GP. I was able to build around these methods in the main method in order to create runs.

5.2.4 Other Methods and Conclusion

During implementation I experimented with various combinations of driving factors. In SNGP the most effective climbing approach I found was to drive on either the increase of the total population fitness or on the fittest members fitness increase;

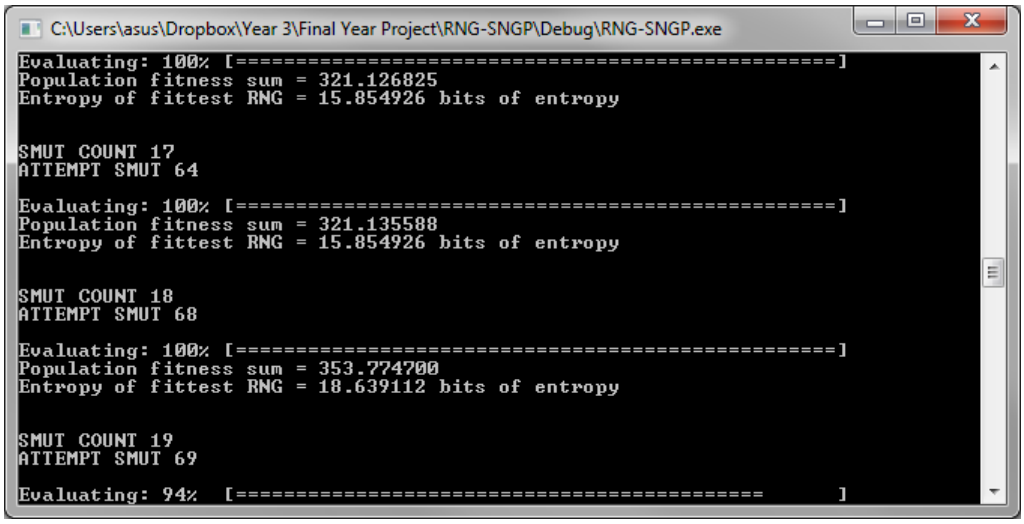
```
if (best_fitness_old >= best_fitness && total_pop_fitness_old >= total_pop_fitness) {
 //Then throw away new graph population after mutation as neither the best fitness
 //or the total population fitness have increased.
}
```

I then implemented the termination criteria using simple variables and conditional statements. Since the design document and during implementation I changed the termination criteria with regards to mutations from the number of actual mutations to the number of attempted mutations. As Jackson describes in [2] this gives a fairer comparison against the number of individuals processed in the GP implementation.

After this, I was then able to start conducting runs and writing test data. The code for writing data is simple and can be found in the appendix along with the rest of the code for the SNGP implementation. It simply writes the run data to a text file in the format mentioned in the design document in the appendix. The files produced are named according to the time at which the run was started.

Having completed these tasks, I had a fully operating SNGP solution ready for testing and evaluation. At this point I was confident about the work I had done, and also about the outcome of the evaluation of SNGP vs GP due to the runs that I had already collected through testing both implementations. Figure 7 below is a screen shot from an example run of the SNGP paradigm.

Figure 5: SNGP Screen Shot



5.3 C *rand*() and TRNG Test Program Implementation

Having implemented both the GP and SNGP paradigms, the next and final implementation stage was to write a piece of software in C that would test both the C RNG and the TRNG in the same way that the RNG trees are tested in GP and SNGP.

I set out knowing that the fitness function from the GP and SNGP implementations would need to be used identically. Therefore all I needed to do for both the C *rand*() function and the TRNG at random.org was to collect their binary outputs and populate an array of length 16834. This array could then be tested in the same way in order to produce a fair test.

5.3.1 C *rand*()

I began by implementing some user definitions. When the program is executed, the user is prompted to decide which method should be assessed. If 'c' is pressed then the user is again prompted for the number of runs they would like to conduct. This was implemented using simple *getch*() and *scanf*() method calls (i.e. get char). Once I had this working, the following method is called;

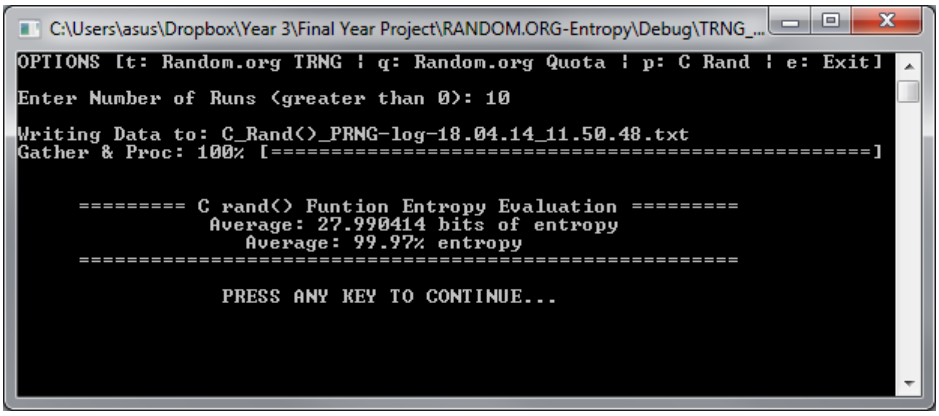
```
void get_c_rand_seq() {
   for (int i = 0; i < J; i++) {
    int bit = rand()%2;
    if (bit == 0)
      bit_seq[i] = '0';
   else
      bit_seq[i] = '1';
   }
}
```

Here I took the remainder of dividing the output of the C *rand*() function by 2 (mod function, so output is 0 or 1), and populated the "bit_seq" character array.

This bit sequence is then assessed using the Shannon entropy equation (from the fitness function) and the program outputs the average (from the user defined number of runs) entropy value and percentage on screen, out of 28 bits (or the max entropy for the defined H_MAX). The run information, consisting of the run number and scalar entropy breakdowns, are then saved into a log file on the hard disk so that it can be used for evaluation purposes.

A screen shot of the C *rand*() implementation can be seen below;

Figure 6: C *rand*() Screen Shot



5.3.2 Random.org TRNG

The other option when the user is prompted is to test the TRNG at Random.org. The same Shannon entropy equation (i.e. fitness function) is used to keep the evaluation of the methods fair.

In order to retrieve a random bit sequence from the random.org servers I had to call the following HTTP request and save the file to disk (deleting the old one from the cache so that we don't get the same sequence twice).

```
void download_bits() {
  HRESULT hr;
  LPCTSTR Url = _T("http://www.random.org/cgi-bin/randbyte?nbytes=2048&format=b.txt"), File
    = _T("RandomBits.txt");
  DeleteUrlCacheEntry(Url);
  hr = URLDownloadToFile (0, Url, File, 0, 0);
}
```

In the HTTP request the "nbytes" parameter is the number of bytes I want to download. As the binary array is 16384 bits long, I want 16384/8 = 2048 bytes. I also want the it to be in bit form (i.e. base 2) and I want it to be text file, hence the format parameter "b" and the final ".txt" extension are added to the request.

Once the file has been downloaded, the binary data is then read into a char array and then the entropy of the sequence is calculated. The code for reading in from the RandomBits.txt file can be seen below.

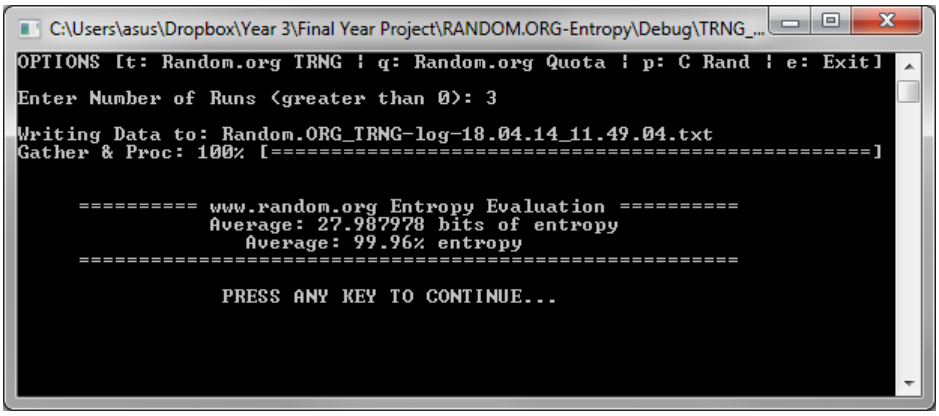
```
for (position; position < J; position++) {
      c = getc(file);

      while (c != '1' && c != '0')
        c = getc(file);
      bit_seq[position] = c;

    }
    //printf("POSITION: %d\n", position);
    bit_seq[J] = '\0';
```

Originally I was reading in all of the sequence except for spaces and '\n' characters. I then noticed that there were other hidden characters I was not compensating for including return line statements (i.e. '\r'). I changed the code so that any character that is not a 1 or a 0 is skipped. As a result I began achieving slightly higher (and correct) entropy values.

Like the C implementation this process is repeated for a user defined number of runs. There is also a check quota function for users to keep track of how many bits are remaining. This is also a text file which is downloaded from the random.org server, and then read in and printed on the screen. The same run data as for the C *rand*() implementation is saved onto the disk for the TRNG data. This entails the run number, the scalar entropy break down and the total entropy.

A screen shot of the TRNG implementation can be seen below;

Figure 7: TRNG Screen Shot



5.4 Conclusion

This concludes the implementation phase of the project. I have now produced three working pieces of software; The Genetic Programming implementation of evolving RNGs, the Single Node Genetic Programming implementation and the implementation of the C RNG, and the random.org TRNG test. All of these implementations output data which shall be collected and used in the evaluation phase of this report.

While there were some hiccups during implementation as can be expected with the development of any piece of software, the process of creating and designing the algorithms and main methods in pseudo code in the design phase was of great use when it came to implementation. I was able to go straight from pseudo code to C code which enabled me to finish all of the implementations a head of schedule (according to the gannt charts present in the design document in the appendix).

6 Evaluation

6.1 Introduction

The first phase of evaluation is to compare the results I obtained from implementing Koza's method of producing randomisers using GP and the results Koza managed to obtain himself. By looking at the data he obtained and what I obtained, we should be able to paint an accurate picture of how well I managed to convey his description in his paper [1].

A major (arguably the most important) part of this project is to evaluate the results produced by both GP and SNGP implementations. The aim is to prove which method of genetically breeding RNGs produces the most solutions in the most efficient manner (with respect to running time and the size of the solutions produced). 50 runs were conducted for both the single threaded GP and SNGP implementations in order to compare averages of their outputs. Runs used for any part of the following analysis where run time is taken into consideration were ran on a machine with an AMD Athlon 64 X2 5600+ 2.8 GHz Dual-Core processor.

The next step of evaluation after that is to compare the RNGs produced by the GP and SNGP paradigms against two other RNGs in wide use today. The C *rand*() function is a PRNG that uses a Linear Congruential Generator algorithm, in order to give developers who are writing C programs a simple way of generating pseudo random numbers. The Hardware/True Random Number Generator at random.org collects binary data from a radio receiver that reads atmospheric noise produced by unpredictable natural effects such as lightning storms. There is an API available where developers can download "true" random data from the random.org servers for use in their software[11]. The randomness/entropy is tested in the same way for all approaches.

Sections of this evaluation have been changed and added since the design document was produced (please see the appendix for the original document). During implementation and initial evaluation, other ways of evaluating the project became apparent to me. I added the evaluation in section 6.2 in order to assess how my results compared to the results presented by Koza in [1]. This way I can tell how accurately my implentation was to that proposed in the research paper. I also added the evaluation in section 6.6, which reviews how well the RNGs produced by Genetic means perform in a real software implementation. The reasoning behind this is that RNGs are frequently used by developers in various programming languages. I wanted to see whether it was practical for developers to use these RNGs produced Genetically in their software, again against the C *rand*() function.

6.2 Evaluation of GP Implementation

The first stage of implementation as discussed in the Realisation seciton of this document was to follow Koza's Paper [1] in implementing a GP to evolve RNGs in C. To do this, I developed algorithms[12] that expressed the processes, equations and design in his paper. The first part of the evaluation is to assess the results that Koza produced against what I managed to acheive from the carbon copy of his implementation.

In section 5.1 of [1, p. 6], Koza introduces his results from the run of the fittest solution produced by the GP implementation. In this run, Koza managed to produce a randomiser with an entropy of 27.996 bits on the 14th Generation. Koza describes the progression in fitness of the fittest candidate over these 14 generations. In order to make a comparison of one of my runs against Koza's run, I must also select the run that produced the fittest solution produced by my implementation. This run of mine took 10 generations to find a RNG with 27.995032 out of 28 bits of entropy.

Koza gives details of his run, saying; "*The best 24 individuals from generation 0 scored between 10.428 and 20.920 bits of entropy. The best single S-expression scored 20.920 bits.*". To compare this to my run, the best prefix expression[13] produced by generation 0 achieved 15.322 out of 28 bits of entropy. Despite the lower fitness in generation 0, the progression of the fitness from therein was similar to that of Koza's run. After the description of generation zero, Koza goes onto explain; "*After 2 generations of one typical run, the entropy of the best-of-generation individual improved to 22.126 bits. After 4 generations, the entropy of the best-of generation individual improved to 26.474 bits. Thereafter, entropy reached and slowly improved within the 27.800 to 27.900 area.*". Again I obtained similar results from my run. After 2 generations the entropy of my best individual improved to 19.105 bits. After Generation 4 it had improved further to 21.005 bits. It improved gradually up until generation 8 where it took a leap from 24.295 to 27.915 bits. This was mainly down to the improvement of the 6 and 7 bit scalar entropies which jumped from 4.994 to 5.986 bits and 5.582 to 6.962 bits respectively between generations 7 and 8. The total entropy of the fittest candidate dropped slightly on generation 9 to 27.811 bits before finally rising to the terminating entropy of 27.995 bits on generation 10.

[11] Please see the implementation section of this document for details on how this is achieved

[12] Found in the Design section of this document and in the design document in the appendix

[13] Koza wrote his implementation in LISP. S-Expressions and Prefix expressions are synonymous, as S-Expression are essentially prefix expressions with parenthesis

Koza also defined the scalar entropy breakdown of his fittest solution; "*In scoring 27.996, this randomiser achieved a maximal value of entropy of 1.000, 2.000, 3.000, 4.000, 5.000, and 6.000 bits for sequences of lengths 1, 2, 3, 4, 5, and 6, respectively, and a near-maximal value of 6.996 for the 128 (27) possible sequences of length 7.*" However, our fittest solution was not so uniform in breakdown. Instead it achieved maximal entropy of 1.000000 bits for sequences of length 1 and then 1.999935, 2.999866, 3.999688, 4.999333, 5.998764, 6.997446 bits for sequences of lengths 2, 3, 4, 5, 6, and 7 respectively. Therefore our solution here is slightly better at randomising sequences of length 7 bits, the same for length 1 bits, but slightly worse for 2, 3, 4, 5, 6 bits. Therefore it achieves a slightly lower overall entropy of 27.995 bits against Koza's 27.996 bits, but both remain highly credible randomisers with 99.98% and 99.99% entropy respectively.

Coincidentally, the solution that I produced was not only the fittest but was also the quickest to find and had the smallest size over the 50 runs. It has only 15 points against Koza's 153 points, and as discussed above, took only 10 generations to find (434.4 sec). The prefix expression can be seen below;

/ - 2 - 2 J % J - 2 / / J 3 3

However the coincidence ends here. The average size of the solutions produced by our GP implementation (across 50 runs) is 154.6 points, very close to the 153 points in Koza's fittest candidate[14] Here we have shown that using one run from both Koza's research in [1] and from the results that I have obtained by replicating his research, that the execution behaviour and the results are very similar.

6.3 50 Standard Runs: GP vs SNGP Implementation

In this section we compare the GP and SNGP implementations in order to uncover which is best suited for the problem domain of genetically breeding RNGs. In order to do this, data from 50 runs of both implementations were collected. These 50 runs were taken under standard conditions for both implementations as seen in the parameter definitions below in table 4. For the GP implementation, these parameters were all taken from there definitions by Koza in [1, p.6]. For the SNGP implementation, the termination fitness was also taken from Koza's definitions. The "Mutation Attempts" and "Population Size" parameters were taken as run conditions for SNGP defined by Jackson in [2, p.54].

Table 4: GP and SNGP Standard Parameters

	GP	SNGP
Max Entropy	27.990 Bits	27.990 Bits
Max Generations / Mutation attempts	51	2500 [15]
Population Size	500	100
Max init program depth	4	N/A
Max crossover program depth	15	N/A
FPR prob	0.1	N/A
Crossover prob	0.9	N/A
Internal node crossover prob	0.9	N/A
External node crossover prob	0.1	N/A

Now that we have defined the parameters, the data from 50 runs of the GP and SNGP implementations can be collected. The format of the data written by the runs can be found in section 4. The raw data written by the 50 runs can be found in the appendix, here this data shall be interpreted in order to conduct evaluation.

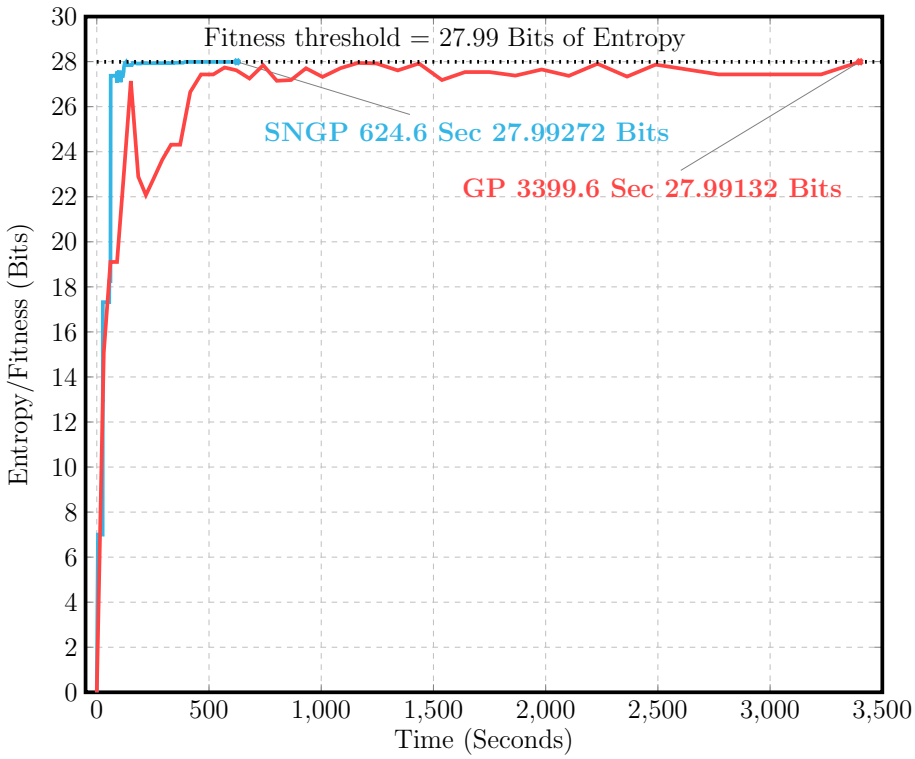
In order to correctly evaluate GP vs SNGP using this newly aquired data, we must first define the criteria which will determine the victor. The aim of any Genetic Program or Genetic Algorithm is to act as a search heuristic in order to find an optimal solution. Therefore like with any search algorithm, the desired characteristics are that it will produce a high rate of optimal solutions in an efficient manner. Applied directly to Evolutionary Algorithms, this corresponds to a process which produces the highest rate of short, highly fit solutions in the quickest time possible.

6.3.1 Single Run Example

Let's first introduce an example of a single run of both the SNGP and GP implementations that both managed to reach the termination entropy. In figure 8 below, we can see already that in this instance SNGP has a more accelerated fitness progression and has a much shorter execution time;

[14] Koza does not provide any average data or information for solution sizes.
[15] Adaptation from 25000 to 2500 explained further on

Figure 8: Sample GP vs SNGP Run



The fluctuation in entropy across the whole run is much less in SNGP than it is in GP. This is reflected when we show the scalar entropy breakdown for the same runs as seen in figures 9 and 10 below. We can see that the SNGP entropy rises in more uniform steps across all 1 - 7 bits, whereas GP fluctuates more aggressively.

Figure 9: Scalar Entropies SNGP

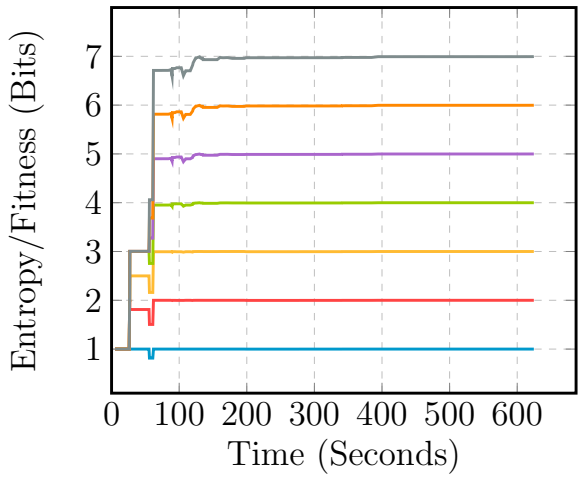
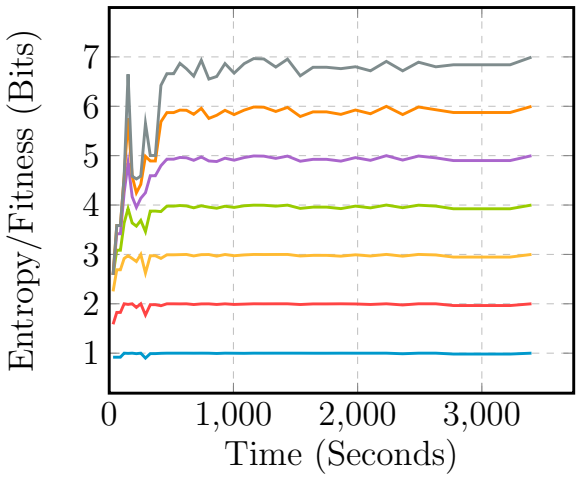
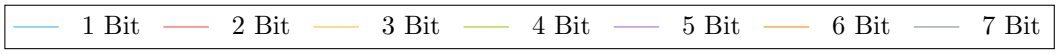


Figure 10: Scalar Entropies GP





The reduced fluctuation for SNGP is down to the fact that it is driven by a hill climbing approach as discussed in both the design and implementation sections of this report. These two runs of each implementation were taken as closest to average for their running time with respect to the other 50 runs of each implementation.

6.3.2 50 Runs: Standard Parameters

After 50 runs of both implementations under standard parameters, the following average results were achieved shown in table 5 below;

Table 5: 50 Run Average: GP vs SNGP

	SNGP	GP
Avg Run Time	606.9s	4111.6s
Avg Sol Run Time	483.2s	2173.4s
Avg Non Sol Run Time	1701.1s	5403.8s
Solution Rate	88%	40%
Avg Sol Size[16]	27.9	154.6
Avg Final Size[16]	30.1	277.8
Avg Run Fitness	27.88153318	27.2570626
Best Sol Fitness	27.995729	27.995032
Avg Sol Fitness	27.9917378	27.99195145

SNGP comes out victorious on all tests bar the Average Solution Fitness (which is only beaten by 0.00076% entropy). Over the 50 runs, SNGP finds over twice as many solutions, in under a quarter of the time. The solutions found are also on average 5 $\frac{1}{2}$ times smaller due to the fact that branches in the trees are repeated (and can be calculated dynamically).

In [2, p.54] Jackson defines that the number of attempted *smut* operations should be based on the number of individuals that would be processed in a traditional GP implementation. This is calculated by taking the product of the max number of generations[17] and the size of the population. Therefore for our corresponding SNGP implementation this should be; $50*500 = 25000$. However Jackson also goes onto say in [2, p.58] "*these can be manipulated to tune the solution rate, solution sizes, and speed of solution discovery*". Therefore for the 50 runs I conducted for SNGP, I reduced the *smut* attempts in order to show both a higher solution rate and a quicker running time. I did this by reducing the parameter from 25000 to 2500 attempts. Table 6 shows 10 runs with the *smut* attempts set to 25000, and it shows that a 100% solution rate can be achieved by doing so. However, taking a look back at our 50 runs, reducing this parameter by a 10^{th} did not reduce the solution rate by the same amount. In fact we can say therefore (loosely) that on average 88% of solutions are found within the first 2500 *smut* attempts.

Table 6: 10 Run Average: GP vs SNGP (25000 *smut* limit)

	SNGP	GP
Avg Run Time	1947.8s	4111.6s
Avg Sol Run Time	1947.8s	2173.4s
Solution Rate	100%	40%
Avg Sol Size	30.8	154.6
Avg Final Size	30.8	277.8
Avg Run Fitness	27.99154673	27.2570626
Best Sol Fitness	27.995936	27.995032
Avg Sol Fitness	27.99154673	27.99195145

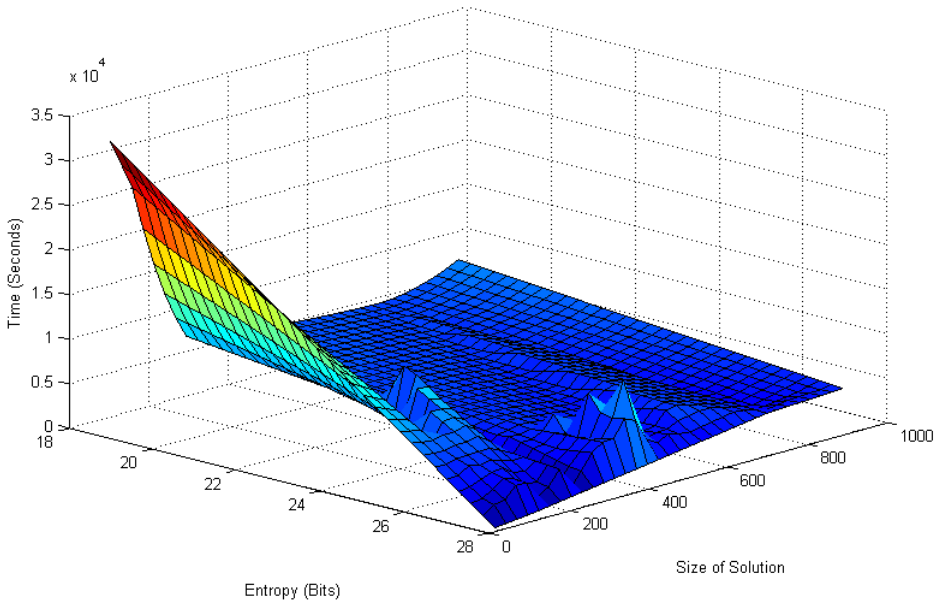
By increasing the number of *smut* attempts from 2500 to 25000 we have obtained a 100% solution rate but at the expense of having an average solution run time close to that of the GP average solution run time.

Therefore we can also say that using 2 independent runs (which can be run in parallel) of the SNGP paradigm given 2500 *smut* attempts and given a success probability of $p_s = 0.88$. That using simple probabilities, the chance of success over those two runs is equal to $1 - (1 - 0.88)^2 = 0.9856$ or 98.56%. Therefore with a reduced *smut* attempt parameter we can achieve a probability of success close to that of 100%[18] and benefit from a reduced amount of computation time.

By taking all 50 runs of each implementation (solutions and non-solutions), we can gain further insight into the relation between solution size, run time and entropy. To do this we can generate 3D surface plots from the data. In figures 11 and 12 below we can see the correlation between solution size, fitness and run time for all 50 runs of both the GP and SNGP implementations respectively. As runs are independent to one another, interpolation was required in order to create a surface in Matlab;

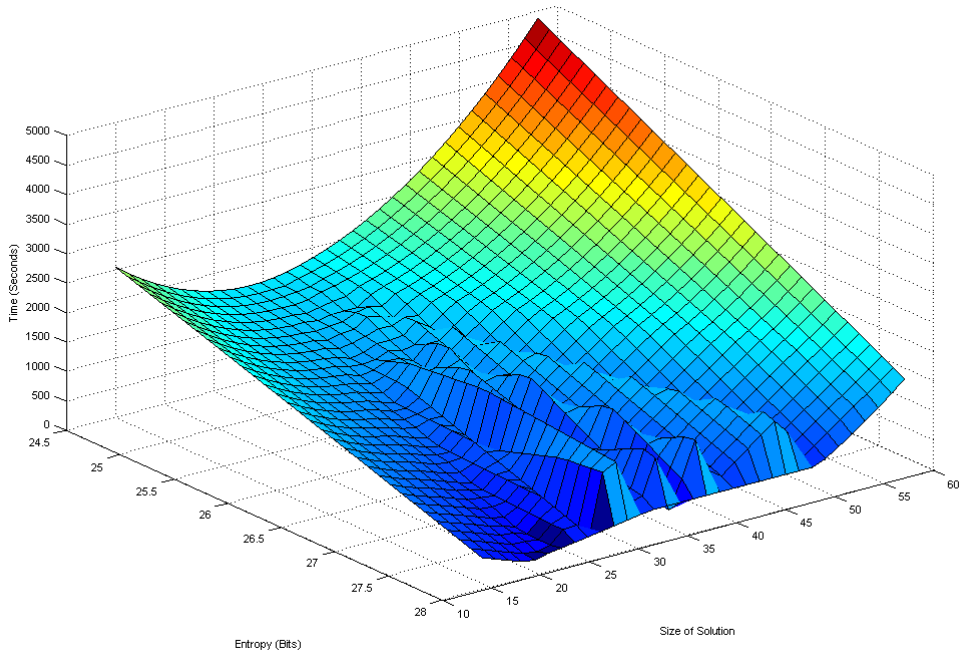
[16] Since the demonstration, solution size fixed for SNGP. It now corresponds to the number of nodes used in a solution.
[17] Not including the initial generation
[18] Getting close but never quite 100% due to the limit of the function ($\lim_{x\to\infty} 1 - (1 - 0.88)^x = 1$)

Figure 11: GP surface plot



In figure 11 above we can see the correlation between between solution size, fitness and run time for all 50 runs of the GP implementation. We can see that the scales are larger than that of the SNGP surface plot in figure 12 for solution size, fitness and run time. There is also less correlation between the runs/points on the plot compared to that of the SNGP plot. This reflects the lower solution rate that GP has, meaning that there aren't groups of points with high fitness plots like that evident in the SNGP plot below;

Figure 12: SNGP surface plot



6.3.3 Fittest RNG Found

After 50 executions of the SNGP and GP implementations, we were able to acheive a RNG with an entropy of 27.995729 out of 28 Bits in the SNGP implementation. The tree can be seen in figure 13 on page 29 below.

The RNG tree in figure 13 can be simplified by pre-calculating branches in the tree that operate only on constant values. When represented as an infix expression in a programming language, compliers will calculate

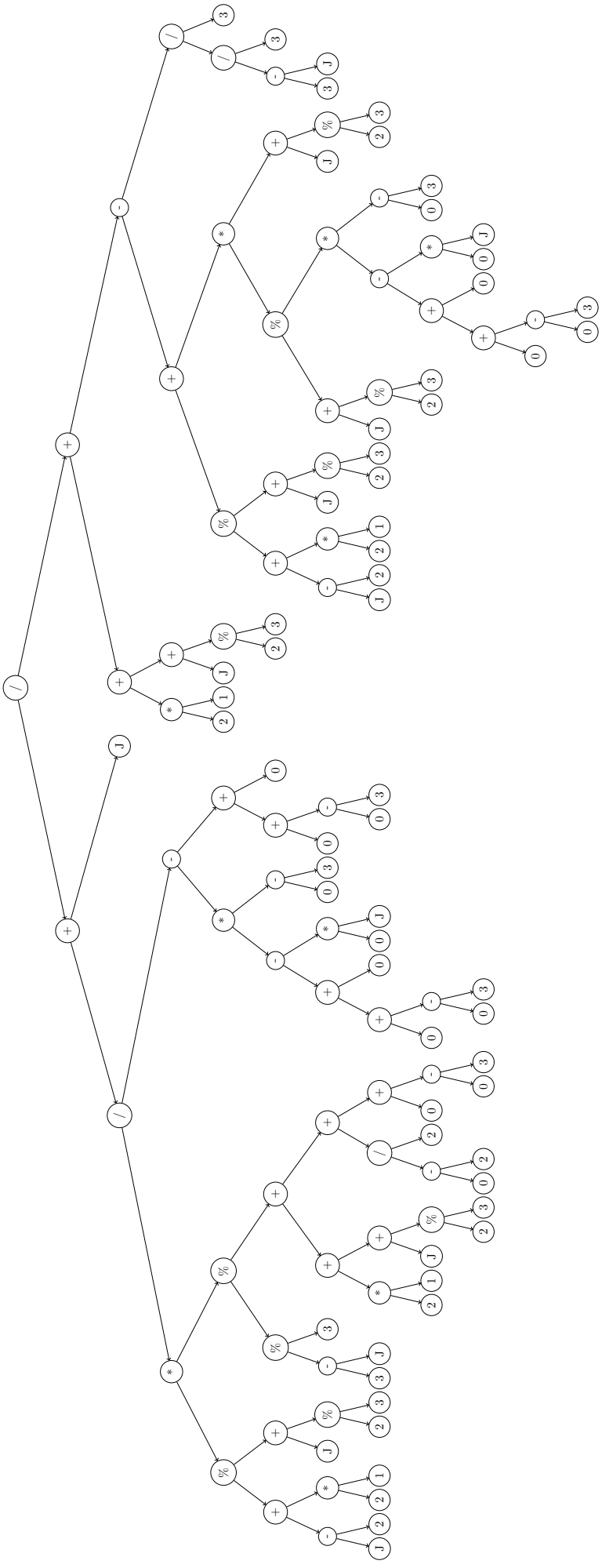
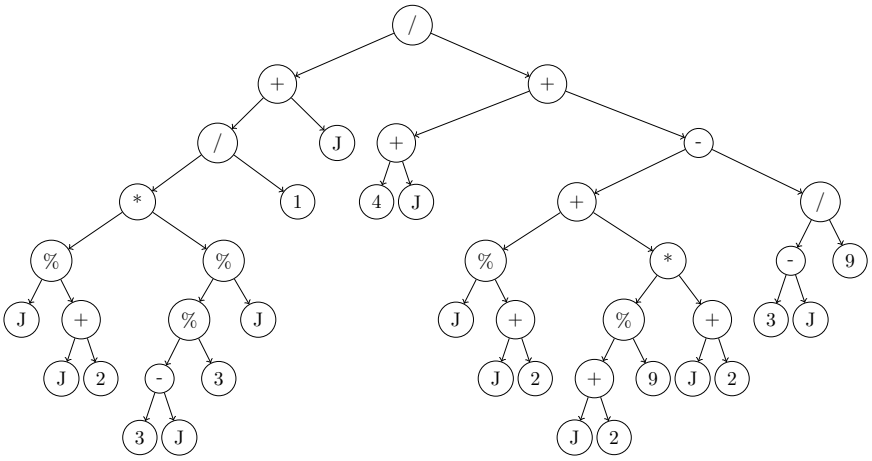


Figure 13: Fittest RNG Tree - 27.995729 Bits of Entropy

this at compile time in order to reduce runtime in what's known as Constant Folding and Partial Redundancy Elimination. The tree in figure 13 can therefore be simplified into the tree in figure 14 below.

Figure 14: Simplified RNG Tree



This simplified tree can be represented as an infix mathematical function $f(J)$ below;

$$f(J) = \frac{\left(\frac{(J \bmod (J+2)) * (((3-J) \bmod 3) \bmod J)}{12}\right) + J}{(4+J) + \left((J \bmod (J+2) + ((J+2) \bmod 9) * (J+2)) - (\frac{3-J}{9})\right)}$$

Such an infix expression can be implemented into a program (i.e. into code). This shall be discussed later in the report.

6.3.4 GP vs SNGP Conclusion

We can conclude that for the purposes of producing RNGs by genetic means, that SNGP is a substantially more efficient and more successful paradigm to implement to solve the problem than GP is. We have also shown that it is possible to manipulate the number of *smut* attempts (increase it 10 fold) to achieve a 100% solution rate.

6.4 Parameter Altering Experimentation: Reducing Population Sizes

In this section we take a look at the effects of altering the population size of the GP and SNGP implementations, in order to see what effect different conditions have on their performance. In [2, p.56] Jackson experiments with the effects of reducing the population size on the solution rate of his 6-mux SNGP implementation. In section 6.3 we took a look at increasing the number *smut* attempts on the performance of the SNGP model. In figures 15 and 16 below we can see the effects of a reduced population on the rate of solutions produced by both methodologies. For SNGP, I gathered data from 10 runs at each of the following 20% population size decrements (and a final 10% decrement) {100, 80, 60, 40, 20, 10}. I gathered the same data from the GP implementation for the populaiton size decrements of {500, 400, 300, 200, 100, 50}. The results I obtained can be seen below;

Figure 15: Reduced Pop size of SNGP

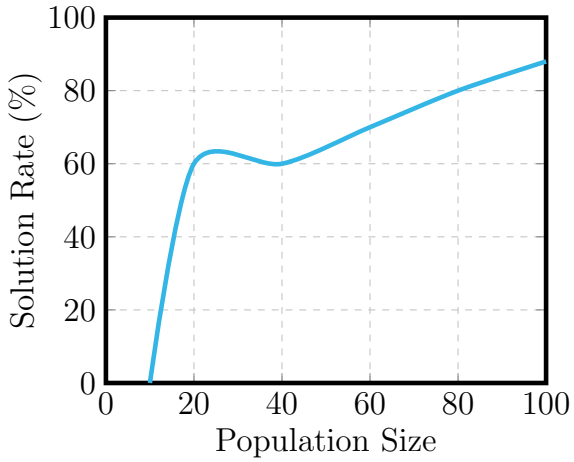
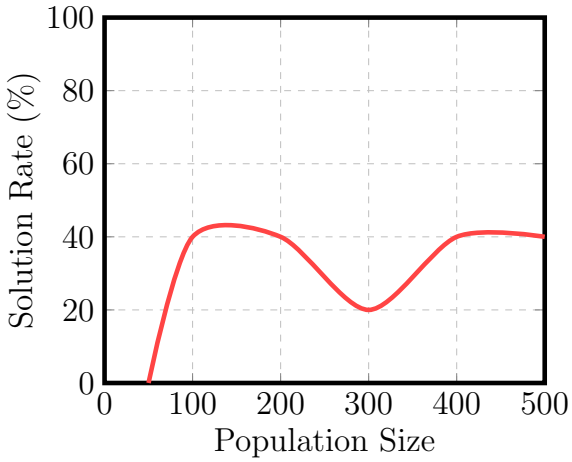


Figure 16: Reduced Pop size of GP



It seems that for both implementations, the solution rate decreases slowly (bar the 20% anomaly for size 300 in GP) until after the population size is set lower than the average size of solutions for the 50 runs as discussed earlier. For GP, lowering the population size past 100 appears to limit the diversity of the population so far that solutions can no longer be genetically bred. For SNGP the average size of solutions is 27.9, and the solution rate for SNGP is reduced to 0% when the population size is set less than or equal to 10. We can see that by limiting population sizes, we limit the diversity of the population and therefore the number of solutions that are available to us.

In figures 17 and 18 below, we can also see the reduction in average running time of both the SNGP and GP implementations when the population size is lowered respectively;

Figure 17: Avg SNGP Run Time with Reduced Pop

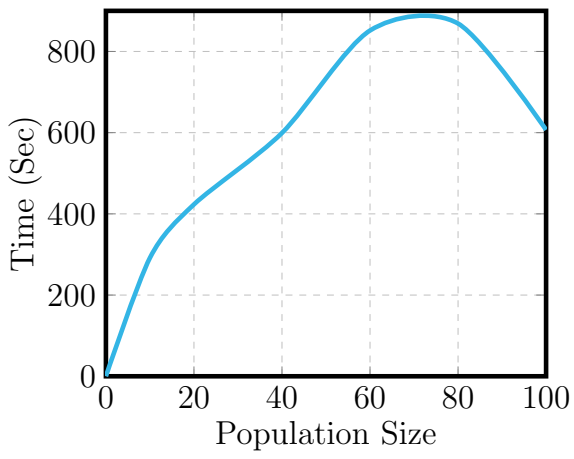
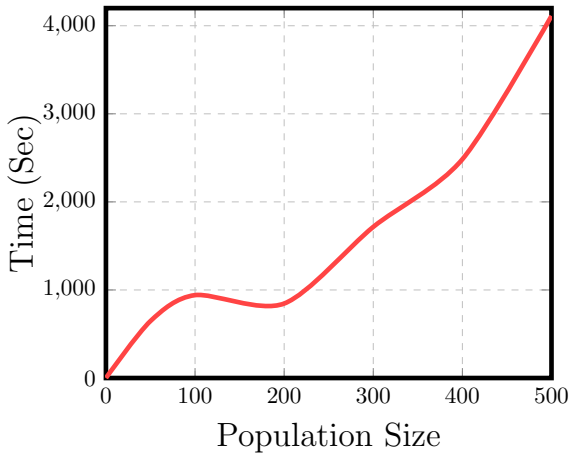


Figure 18: Avg GP Run Time with Reduced Pop



In figure 18 above we see the expected reduction in run time as we reduce the population size. This is because the number of individuals being processed is being reduced and therefore the number of trees that must be evaluated and tested for fitness is also reduced. However in figure 17 above we see that the average computation time actually rises from 606 to 868 seconds when the population size is reduced from 100 to

80, and doesn't actually drop back below 606 seconds until a population of size 40. This can be seen as an anomaly (as only 10 runs were taken for each population size). However it could be reasoned that we know (from our 50 runs in figure 5 on page 27) that runs that fail to find a solution take on average 3.5 (1701.1/483.2 = 3.521) times longer to find a terminate than those that do. Therefore if the solution rate drops from 88% to 80%, this will have a knock on effect to the average running time for not finding solutions (minus the reduced running time for processing less individuals). For instance, when the solution rate levels off at 60% between 40 and 20 the run time decreases, as a reduced population size with the same solution rate will definitely incur a reduced run time.

Here we have shown that a reduced population size can reduce the run time of both implementations whilst still yielding an acceptable solution rate. However we have also shown that (in the case of SNGP), a reduced population size that reduces the solution rate, can lead to non solution times having a more aggressive effect on the average run time, than the reduction in time that having less individuals to process does.

6.5 Genetic vs Conventional RNGs

Now that we have determined that SNGP is substantially better suited for the actual process of generating of RNGs, we can now use the RNGs in order to compare them against other conventional RNGs. As discussed earlier in this report, I have chosen two different random number generators as competitors to our genetically bred solutions. I chose the C *rand*() function incorporated in the standard C programming language library as a comparable PRNG that (like our RNGs) produces numbers deterministically. Unlike our RNGs which use a sequence of integers, the C *rand*() function uses a Linear Congruential Generator (LCG) algorithm which uses a seed value in it's equation in order to output a number which is then used as the seed for the next call. This LCG algorithm/equation can be seen below;

$$X_{n+1} = (aX_n + c) \mod m$$

where;
$m, 0 < m$ - the modulus
$a, 0 < a < m$ - the multiplier
$c, 0 \leq c < m$ - the increment
$X_0, 0 \leq X_0 < m$ - the seed

As shown in [13], LCGs are vulnerable to attack. In the paper it is shown that the variable m in the equation above can be calculated given only 3 outputs due to how LCG outputs commonly fall into 3 dimensional planes. A possibility with using RNGs that are evolved genetically, is that their formula would not be widely known and the formula itself can be updated periodically by the implementer by generating new RNGs.

For the other comparable RNG, I chose a Hardware / True Random Number Generator. This way we can compare our genetically bred randomisers to a source of non-deterministic random data (i.e. the random values are not the result of some deterministic algorithm or equation). I chose to specifically use the service provided at random.org [15], where their servers provide random binary digits generated from observed atmospheric noise on demand.

We analyse the random data produced by these two methods in the same way as our fitness function in our Genetically bred RNGs. Using the Shannon entropy equation we can gather 16834 bits from the other two RNGs and calculate the entropy for binary subsequences of lengths 1 to 7 bits. This way we can test all of our RNGs fairly under the same conditions and using the same test.

I used the data from the 50 runs of our GP and SNGP implementations, and then ran the same number of tests on the conventional RNGs. The results can be seen in table 7 below;

Table 7: TRNG vs C PRNG vs SNGP vs GP

	TRNG[19]	C PRNG	SNGP	GP
Fittest (Entropy)	27.993975	27.991925	27.995729	27.995032
Fittest (%)	99.978%	99.971%	99.985%	99.982%
Fittest (Entropy Shortfall)	0.006025	0.008075	0.004271	0.004968
Avg Fit (Entropy) (Sol)	27.98997218	27.98985372	27.9917378	27.99195145
Avg Fit (%) (Sol)	99.9641%	99.9637%	99.970%	99.971%
Avg Fit (Shortfall) (Sol)	0.01002782	0.01014628	0.0082622	0.00804855

From the results shown above, we can see that the RNGs produced by Genetic means outperform both the C *rand*() function and the TRNG in terms of entropy. SNGP produces the candidate with the highest fitness and GP produces the highest average fitness (slightly higher than that of SNGP which also beats the

[19] Fixed since demonstration (The test code was including some formatting characters and not just binary data)

two conventional methods). Whilst the genetic RNGs only beat the conventional methods by a fraction, this small improvement can make all the difference when producing high entropy random digits.

The C *rand*() function performs the worst out of all four candidates producing both the lowest best run and the lowest average fitness (albeit very marginally to the TRNG). The TRNG then places 3^{rd} producing lower average fitness and lower best run fitness than both of our evolutionary methods.

Random data can be interpreted in different mediums such as in audio, video and images. We can create 128x128 ($\sqrt{16384} = 128$) images for each of our random methods by interpreting a single bit as a black pixel if it is equal to 1 or as a white pixel if it is equal to 0 from the sequence of 16834 bits.

The images in figures 19, 20 and 21 were created from 3 runs chosen from each of the methods;

Figure 19: C *rand*() Image

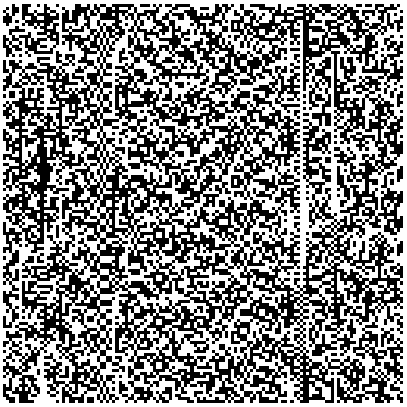


Figure 20: Random.org Image

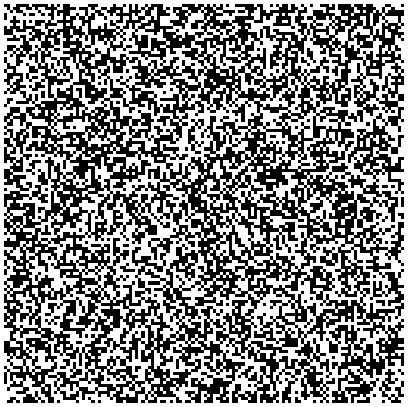
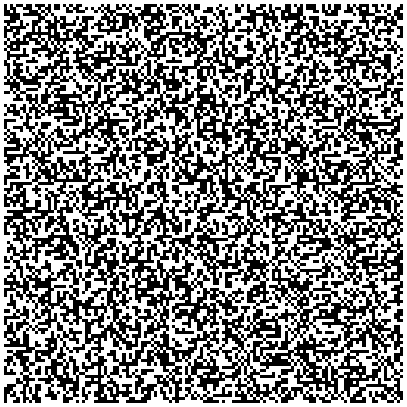


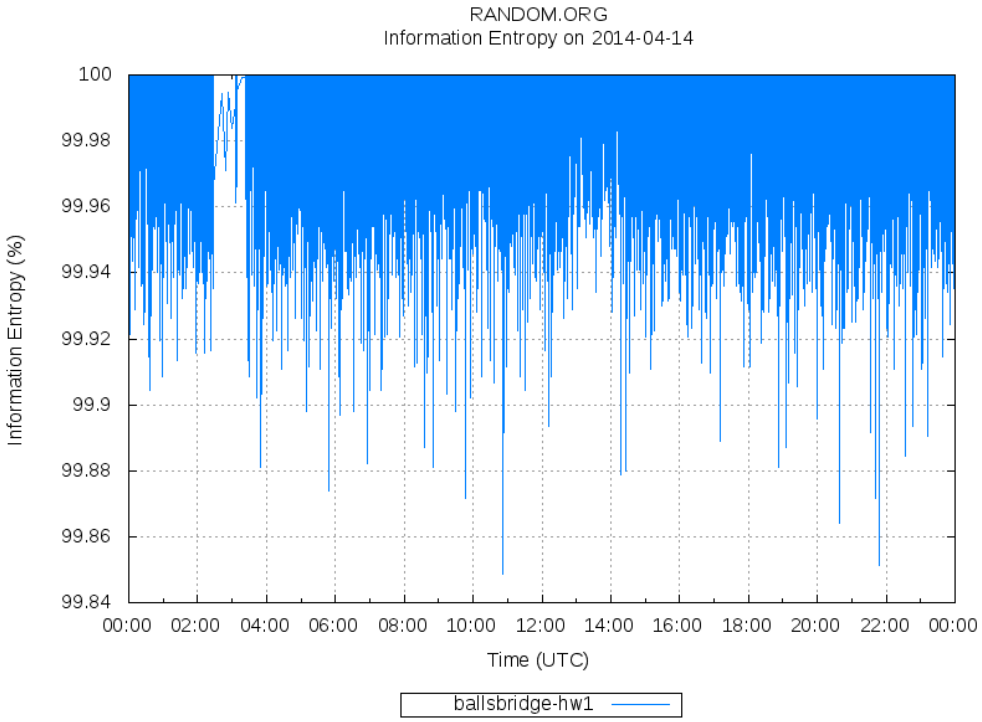
Figure 21: Genetic Image



As we can see the C *rand*() function displays repetitions at uniform intervals giving rise to verticall patterns (the images are populated with pixels from to left to top right and then repeated down the image). This shows that numbers (binary sequences) are repeated at uniform intervals as well as more often (giving rise to lower entropy). The data from random.org has no distinguishable patterns at all, however this does not necessarily correspond to a higher entropy, as numbers may still be repeated just not in a uniform fashion like that of the C *rand*() function. On close inspection the data/image from the genetic RNG might show a subtle pattern to some viewers eyes. This is in fact what gives rise to its very high entropy; a uniformity in distribution and a lower repetition. As our genetic RNG has the highest entropy, in these 16384 bits/pixels (128x128) every binary subsequence of length 1 to 7 needs to occur with (almost) equal probability. Therefore when the GP and SNGP are genetically "engineering" this through evolution, it causes RNGs to produce some uniformity.

Below is a graph produced by the random.org servers displaying the shortfall in entropy on the same day on which I collected my test data. As the website provides no specific details on the size and length of bit sequences on which the output from their TRNG is tested, the use of this graph isn't a fair comparison against our other RNGs. However it seems that the average is around 99.96% to 99.7% which is close to the results we obtained. As we can see from the graph, as the source of the TRNG is non deterministic, the entropy can fluctuate quite aggressively;

Figure 22: Random.org Entropy Shortfall



For comparison, the shortfall in entropy of SNGP and GP solutions can be seen in figures 23 and 24 below.

Figure 23: SNGP Entropy Shortfall of Solutions

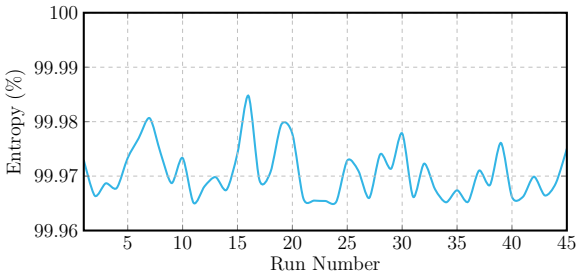
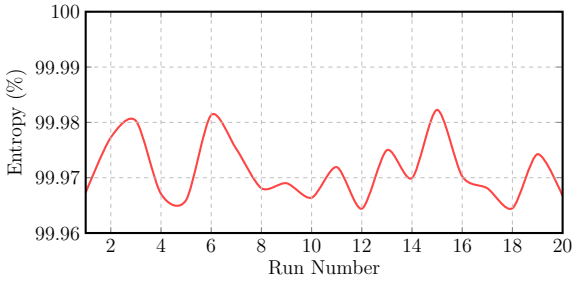


Figure 24: GP Entropy Shortfall of Solutions



To conclude, statistically the RNGs produced by evolutionary means outperform the two common conventional methods as far as we are concerned with entropy. To someone else however the data in the image produced by the TRNG may be deemed more "trustworthy" or more random due to its non-uniformity in distribution. As far as PRNGs go however, both visually and statistically our RNGs produced by evolutionary means are evidently superior to the C *rand*() function.

6.6 Real Implementation

Now that we have shown the RNGs produced by genetic means are superior (in terms of entropy) to widely used conventional methods of producing random numbers, another way to assess the successfulness of this project is to see their real world application. Take the fittest and smallest solution produced by the GP implementation as discussed in section 5.2;

$$/ - 2 - 2 \text{ J } \% \text{ J } - 2 / / \text{ J } 3 \text{ } 3$$

This RNG achieved 27.995032 out of 28 bits of entropy. It can be represented as an infix mathematical function like so;

$$rand(J) = \frac{(2 - (2 - J))}{J \mod (2 - (\frac{J}{6}))}$$

Remember that as discussed before, the RNGs are evolved on their binary output entropy and not as integer interpretation. Therefore in order to obtain integers with high entropy in an implementation, the RNG must create a stream of binary digits to then be interpreted as integers. They are also evolved based on their ability to randomise a sequence and not as a LCG. Therefore, in an implementation a seed value should be attained and then incremented in order to produce the best results. An example implementation of the function above can then be implemented in C code as seen below.

```
int randomiser(int min, int max) {
  max++; //Incrementing Max as to include it as a possible random output.
  int dec = 0;
  //Taking into account a negative minimum
  if (min < 0)
    max = (-1*min) + max;
  //Calculating the length of the stream we need
  int bit_max = (int) ceil(log10((double)max)/log10((double)2));
  //For that length we can read in the binary output of the random number generator
  for (int i = 0; i < bit_max; i++) {
    if ((protected_div(2 - (2 - seed), protected_mod(seed, (2 - (protected_div(
    protected_div(seed, 3), 3))))) % 2) == 1)
      dec = dec + pow((double)2, i);
    seed++; //Running through the sequence as we do.
  }
  //Returning the value between the defined range, taking into account a negative min
  int rand_return = 0;
  if (min < 0)
    rand_return = (dec % max) - (-1*min);
  else
    rand_return = ((dec % (max-min)) + min);
  return rand_return;
}
```

As discussed earlier the protected division and protected modulo functions return 1 if division or modulo by 0 is attempted. This function can be seeded and called like so;

```
int seed = time(NULL); //Seeded with the current time
int rand = randomiser(-300, 2000); //Generate a random number between -300 and 2000
```

Calling the randomiser function 100 times using the same parameters as seen above gave the following output;

```
637  886  136  605  1256 393  555  213  969  715  290  1813 1429 557  26   1317
1369 1441 157  -169 1668 689  -8   1028 429  1106 24   665  847  160  940  -144
-237 -215 1138 -235 286  1007 -86  1266 -225 1111 1518 -62  1023 984  1218 1235
1881 1788 1201 1903 -294 1244 1375 1258 380  1624 167  440  -141 1956 557  -273
778  495  1047 1288 706  1323 1908 1594 1076 665  1261 31   1432 1397 166  279
975  1195 1758 1084 932  658  84   408  400  168  420  337  72   -222 561  -122
485  -16  -154 792
```

Using this method, while producing a higher entropy result, is computationally more intensive than the C rand() function. This is because the RNG must be called n times where n is the number of bits required to reach the *max* value defined by the user. This RNG can easily be parallelised however, so each RNG call can be done in parallel (because we can predict the seed values in advance). As we can see in figure 25, the running time of both the C rand() function and our genetically evolved RNG with respect to the number of calls on both are of the same order. However there exists some constant number of operations in our RNG implementation, that causes a longer over all running time. We can also see in figure 26 that over 1000 runs of each input, a higher value has no effect on the C *rand*() function, as the limit of the number is done with a simple modulus function. With our implementation of the genetically bred RNG, the max input size dictates how many bits we need and therefore how many calls of the RNG we need. Hence a larger max value has an effect on the running time (albeit small). We can conclude therefore that if an implementation of our RNG was desired by a developer, then a higher entropy output would come at the expense of computation time.

Figure 25: Logarithmic (base 10) Scale Run time of n calls of $rand(J) = \frac{(2-(2-J))}{J \mod (2-(\frac{J}{6}))}$ and C $rand()$, on the fixed min value of 0 and max value of 2000 for all runs

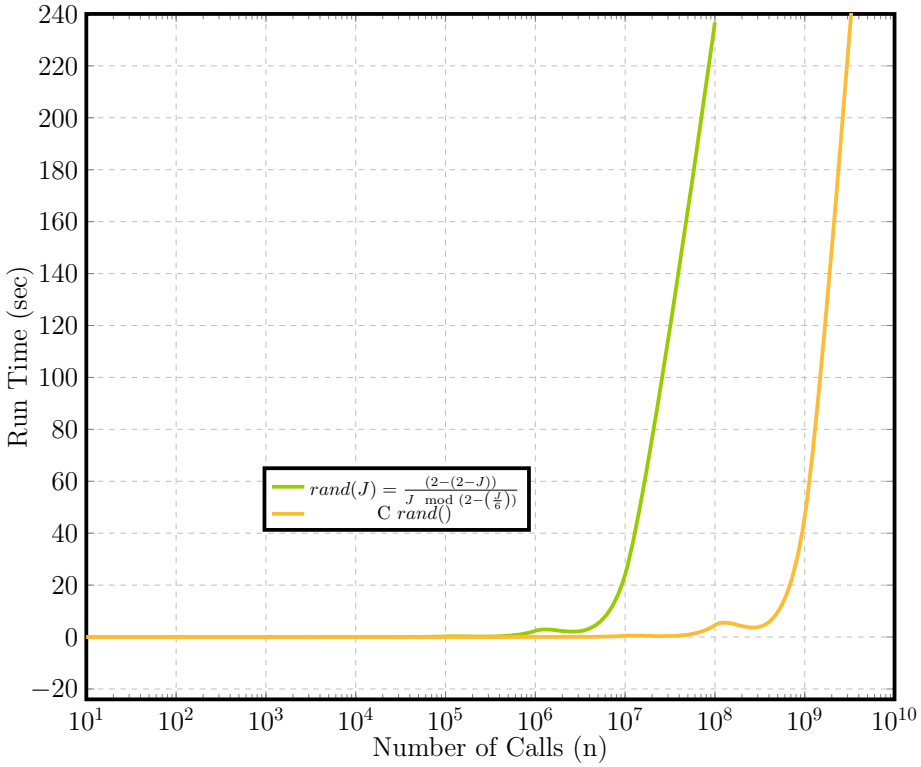
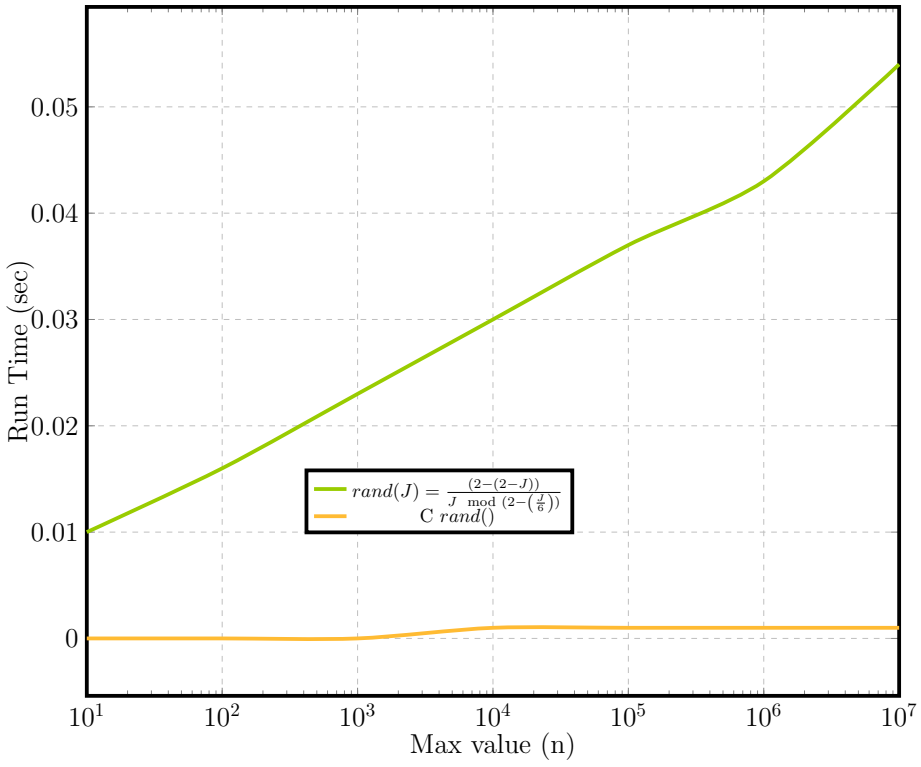


Figure 26: Effect of increasing max value of random number on run time, 1000 calls for each max value. Logarithmic (base 10) Scale Run time of $rand(J) = \frac{(2-(2-J))}{J \mod (2-(\frac{J}{6}))}$ and C $rand()$



7 Discussion

Having completed this project, I leave open further research that can be conducted. A multi objective optimisation implementation for GP and SNGP may be implemented in order to produce RNGs which are not only capable of producing high entropy, but also perform well in other statistical tests for randomness covered in the NIST test suite [3]. Of course, there would most likely be some amount of trade off between multiple statistical results, so the termination entropy may need to be lowered in order to obtain results.

Further mathematical operators could be added to the set of functions. By taking x as the value of the left branch and y the value of the right branch, any mathematical operation with an arity of two might be used. This could include the exponent function (x^y - although this is a computationally intensive function) or the logarithmic function (with either a coefficient and a static base i.e. $x \log_2 y$, or a dynamic logarithmic variables i.e. $\log_x y$). The x^{th} root function could also be used, in the form $\sqrt[x]{y}$ where again, x and y are the values of the two branches of the node.

These operations may lead to larger computation times against the simple mathematic operators used in this project, but they may also lead to fitter solutions.

8 Learning Points

In Summer 2013 after selecting this project I began doing preliminary reading around the areas of Genetic Algorithms and Genetic Programming. Before this time, I had read only briefly around what GP and GAs were capable of solving and their basic structure. I chose this project for two reasons; one because I thought that this was a very interesting application of Genetic Programming, and the second was because I wanted to further broaden my knowledge in the area of Computer Science instead of doing a project on something I already had a lot of prior knowledge of. Therefore this project has not only been a success in proving that SNGP is a more effective method for evolving RNGs and Genetic RNGs can perform better than other widely used RNGs, but it has also been a great success in educating me in an area which I also wish to continue to do further research in.

Initially I began reading around the area of Genetic Algorithms, crucially using the book "*An Introduction to Genetic Algorithms*" by Melaine Mitchell [4], which gave me a solid introduction into the world of evolutionary computation. I then refined my reading towards Genetic Programming. Whilst [4] covers Genetic Programming breifly in the book, I moved onto a more detailed and focused book entitled "*Genetic Programming. On the Programming of Computers by Means of Natural Selection*" by John Koza [7]. John Koza is a pioneer of Genetic Programming, and his work on evolving RNGs using Genetic Programming in [1] was the basis for this project (along with Jacksons research on SNGP). From all of this reading and research I gained a solid understanding of the area of Evolutionary Computing and more specifically Genetic Programming.

Having written all of the main algorithms and pseudo-code for the GP and SNGP models in the design section, the process of implementation went much smoother than I had first expected. I spent the majority of the time during implementation simply translating the algorithms from pseudocode to C code. There were fewer bugs and semantic errors than I had anticipated, and as a result I was able to finish implementation ahead of schedule. From this process I was able to learn that thorough design can save a lot of time during implementation. From past projects (academic and non-academic) where design was not as extensive, I had spent longer in the implementation stage. In the future for any projects and research I conduct, I now know that a through design can save me time and make implementation simpler to translate from design to code.

This project has also further improved my skills in research. My ability to use scientific papers related to the work I am conducting, in order to further build upon and find new ways of comparing results has vastly improved in the duration of this project. A large section of this project was down to analysing and evaluating the performances of all methods of producing random numbers. My analysis techniques have also been improved during this project, I have developed better approaches to testing thoroughly and well and hope to carry what I have learnt into research projects in the future.

From a technical standpoint, I now have a better mathematical understanding of information theory and entropy, and also improved C programming abilities especially when implementing the Shannon entropy equation in the fitness function. I have improved my ability to create efficient algorithms and analysing code for bottle necks (such as identifying the need for the KMP pattern matching algorithm in my fitness function). Seeing as all reports, presentations and documents for this project are written in LaTeX, my skills with using and coding in the document preparation software have vastly improved. I educated myself in being able to implement all of the graphs (bar the 3D surface plots created in MatLab) that I required for analysis, and I even wrote a small C program in order to covert RNG prefix expressions produced by my GP and SNGP implementations into LaTeX tikz code for drawing trees. I believe my better understanding and capabilities of using LaTeX will be very useful when writing scientific papers or documents in the future.

9 Professional Issues

The software produced as part of this project conforms (where applicable) to the British Computer Society: Chartered Institute for IT's code of conduct and code of practice.

During programming, I conformed to the code of "*Strive to produce well-structured code that facilitates testing and maintenance.*". As seen in both the design and implementation sections of this document and the design document in the appendix, the code I implemented was taken from the well structured and defined algorithms created in the design. Each method encapsulates a specific job and the main method is left uncluttered by conducting method calls in the correct order, copying the traditional Genetic Programming template.

My project also conforms to the code of; "*Produce code that other programmers will find easy to maintain; use meaningful naming conventions and avoid overly complex programming techniques, where these are not strictly necessary*". Other researchers looking to adapt or modify my code, need only look to this document for full details on all of the specific operations of the programs. Variables and structures in the code are named after their theoretical counterparts. For example in the SNGP implementation the array of node structs which represents a graph population is named "graph" and each node is called named "population_member". This will give theoretical computer scientists in the area of evolutionary computing an easy association of the code to their equivalent theoretical objects and structures.

In section 2 of the codes of practice, under "*Manage Your Workload Efficiently*", I believe I met all of these conditions for my project. I managed to finish before deadlines during implementation, and also managed to add extra evaluation to this report. I believe that this was possible because I met all of the requirements for time keeping. Looking at this section in the code of practice and comparing to my own work; No over runs occurred during my project, so none needed to be reported. Extra work was added on but it was done with consideration of time and it was all finished on schedule. By preparing largely for the project in the design phase, all the resources I needed were gathered then and there, and all of the pseudo code was written ready to be implemented.

In section 3 under "*Planning*", I met these criteria by creating extensive specification and design documents. Gannt charts were used as a good visualisation tool for quantifying the remaining work to be done and the time scale on which to do it. The deliverables were agreed with my project supervisor, Dr Jackson before beginning. Koza's paper [1] ticks the box for a similar project, and after reviewing the results he obtained, I knew that I could go ahead and strive for the same results before moving on to the SNGP implementation. Continuing to create Gannt charts throughout all stages of the development allowed me to keep track of the time scale which the project was on and the amount of work I had left to do. This therefore meets the code of practice of "*Tracking Progress*" in section 3.

Under "*When closing a project*" in section 3 of the BCS code of practice; describing the changes I made in the design and implementation and why I made them in this document as well as any difficulties I ran in to meets the first requirement of "*Honestly summarise the mistakes made, good fortune encountered and lessons learned.*". In section 7 I also talk about what research could be conducted next and ways in which this project could be changed. This meets the second requirement; "*Recommend changes that will be of benefit to later projects.*"

After reviewing the BSC code of conduct, neither myself nor the software that I have produced have infringed any of the terms.

10 Bibliography

References

[1] John R. Koza, *Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm*. Stanford University, 1991.

[2] David Jackson, *A New, Node-Focused Model for Genetic Programming*. Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, Springer Verlag 2012.

[3] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel,David Banks, Alan Heckert, James Dray, San Vo *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST, 2010.

[4] Melanie Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1999.

[5] Phillip A. Laplante, *Biocomputing*. Nova Science Publishers Inc, 2003.

[6] R. Poli, W. B. Langdon, N. F. McPhee, J. R. Koza, *A Field Guide to Genetic Programming*. University of Essex, 2008.

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

[8] John R. Koza, *A Hierarchical Approach to Learning the Boolean Multiplexer Function* Stanford University, 1990.

[9] Brian W. Kernighan, Dennis M. Ritchie, *C Programming Language*. 2nd Edition, Prentice Hall Professional Technical Reference, 1988.

[10] David Jackson, *Single Node Genetic Programming on Problems with Side Effects*. University of Liverpool, 2012.

[11] C. Lamenca-Martinez, J.C. Hernandez-Castro, J.M. Estevez-Tapiador, and A. Ribagorda *Lamar: A New Pseudorandom Number Generator Evolved by means of Genetic Programming*. University of Madrid, 2006.

[12] Robert M. Gray *Entropy and Information Theory*. First Edition (Corrected), Stanford Unviersity 2000.

[13] Haldir[RET], *How to crack a Linear Congruential Generator*. Reverse Engineering Team, 2004.

[14] T. Symul, S. M. Assad, P. K. Lam, *Real time demonstration of high bitrate quantum random number generation with coherent laser light* Centre for Quantum Computation and Communication Technology, Department of Quantum Science, Australian National University. 2011.

[15] Haahr, M. *RANDOM.ORG - True Random Number Service. [online] Random.org*. Available at: http://www.random.org [Accessed 28 Apr. 2014]

11 Appendices

Provided with this report is a CD containing; all of the Microsoft Windows Visual Studio C code projects produced for this project, all of the documents produced for this project (a long with their corresponding tex files), and all of the test data gathered that was used in the evaluation of this project. Below is the folder tree structure for the CD containing all of these items.

- Appendix
  - LaTeX Documents ..... LaTeX documents including resources used in them
    - Demonstration
    - Design and Presentation
    - Dissertation
    - Interim
    - Specification
  - Project Documents ..... Compiled LaTeX pdf files
    - demonstration.pdf
    - design-document.pdf
    - interim-report.pdf
    - presentation.pdf
    - specification.pdf
  - Code ..... MS Visual Studio projects and code
    - Data Proc
    - RNG-GP-Koza
    - RNG-SNGP
    - TRNG_C
  - Software ..... Standalone executable applications
    - RNG-GP-Koza.exe
    - RNG-SNGP.exe
    - TRNG_C.exe
  - Test Data ..... Test data gathered and used in the evaluation stage
    - 10-SNGP-25000
    - Demonstration
    - 50-C-PRNG
    - 50-GP
    - 50-SNGP
    - 50-TRNG
    - C-vs-MyRand
    - Reduced Pop
    - SNGP-Fixed-Size

This appendix is also available at; https://github.com/philleonard/Random-Number-Generation-Using-Genetic-Programming

The appendix can be viewed here or it can be cloned (downloaded) In order to clone the repository; open git bash (download it from http://git-scm.com/downloads), and move to the desired folder. Then type;

```
git clone https://github.com/philleonard/Random-Number-Generation-Using-Genetic-Programming.git
```

The repository will then be cloned to your computer.

Once the appendix has been obtained (either from the CD-RW or the GitHub repository), then in order to use the standalone software, navigate to the Software folder and simply run an implementation by opening the application. In this folder there is the GP, SNGP and TRNG/C implementations available. However in ordert o change parameters and change and view the code, then you will need to have a copy of Microsoft Visual Studio installed. The software was written on the 2012 platform. Simply start Microsoft Visual Studios and navigate to the Code directory in the repository/CD. The MS Visual Studio projects can be found here for all three implementations plus an extra program used for post-processing some data used in this report. To open any of them, simply open the Microsoft Visual Studios solution file in any of the directories.

In order to just view the code outside of MS Visual studios, open the folder corresponding to the solution name. In there, for each solution there will be a single C++ (written in C) file, containing all of the code written for each implementation.

As far as the documents go, the pdfs can be found in the "Project Documents" folder in the root of the repository/CD and the LaTeX documents can be found in the "LaTeX Documents" folder.

The data collected for all of the evaluation in this project can be found in the "Test Data" folder in the root of the repository/CD. All data is represented in log files and Is in a Comma Separated Values format.