Random Number Generation Using Genetic Programming

BSc Computer Science Dissertation

by

Philip Leonard

April 14, 2014

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 things I found when evaluating the project.

This report demonstrates that using methods described by Koza in [1], it is possible to genetically breed Random Number Generators that produce sequences of pseudo random bits with near maximal entropy. This report more importantly shows that it is possible to produce better results using the Single Node Genetic Programming methodology described by Jackson in [2]. 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 Linear Congruent Generator 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.

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 Congruent Generator (LCG).

Contents

1 Introduction

Random Number Generators have many computational, scientific and societal applications. They are essential in; gambling, lottery draws, cryptography, statistical sampling, Monte Carlo simulations and other applications where and 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[1] "*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[2]), or they use a sequence of consecutive numbers as an input. Mathematical operations 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 radiation, measured by a Geiger Counter.

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. Inspired by Evolutionary Algorithms, Genetic Programming is a method used to evolve computer programs using the Darwinist theory of natural selection (survival of the fittest). The general framework for Genetic Programming is;

1) To begin with, a random initial population of computer programs (traditionally in a tree structure) is generated from a set of functions/operators and terminals.

2) A fitness function is used to assess the fitness of each program, i.e. how well the program does it's job, or how accurate the computed output is.

3) Genetic operations such as crossover and mutation are used in order to generate new offspring programs.

4) The fitness of the new population is then assessed again, and the programs for the next generation are then selected.

The process from steps three and four are then repeated, creating new generations of "fitter" programs. Generally, Genetic Program runs terminate when a member of the population reaches a certain pre-defined fitness, or when the run has reached a certain number of generations. Fitness functions, probabilities and genetic operations all vary from problem to problem.

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 random number generator 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 Random Number Generators 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 Pseudo Random Number Generator which is makes use of a Linear Congruent Generator algorithm in order to produce random numbers, and a "True" Random Number Generator 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

[1] National Institute of Standards and Technology

[2] 00:00:00 (UTC), Thursday, 1 January 1970

information and run progress to the user, and all data for each generation and each run, including entropy values and 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 to collect 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 funtion and the TRNG.

The effecitvness 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 succesfull 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 wether they stand up to other widely used RNGs (both deterministic and non-deterministic). The results of 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 can be stored and therefore only calculated once and stored dynamically. The solutions produced by the two genetic methodologies also outperformed the C rand function and even the "True" Random Number Generator.

2 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 to support 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 reults 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 challanges of the project. I was able to acheive individuals 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 of the GP implementation struggling to obtain the last 0.1 bits of entropy, and in fact is where most runs spent a large number of generations searching. I also obtained similar sized solutions Koza's results.

As discussed in the specification and design documents, research was the dominating factor in preparing for this project. I began researching 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 implementations of Multiplexer Problem described in [8]. The focal point of my research has been around the papers by Koza [1], which is the inspiration for the implementation of this project using GP, and Jackson [10]. In [1], Koza conducts some tests of the genetically produced RNGs against RNGs that were widely used at the time[3]. 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 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. Seeing 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;

[3]Koza's paper was published in 1991

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 [10], 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.

3 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.

3.1 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[4] 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;

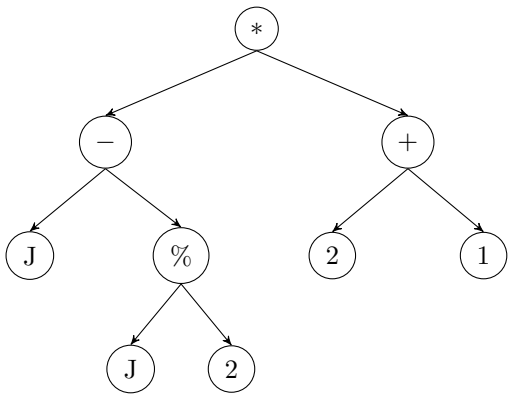
$$T = \{J,\ \Re\}$$

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 figure 2 below;

Figure 2: 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.

[4]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. A 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 entopy of a sequence of binary digits is the equality in the occurrence of a set of binary subsequences of length h in that sequence. The Shannon entropy calculation for all possible 2^h binary subsequences of length h is;

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

Here j ranges over all possible subsequences 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 3 is a simple flow chart representation for the GP. At this point we have defined the first 3 operations. The 4th step is a Boolean decision which determines the continuation or termination of the GP, based on one or both of the two following termination criteria being satisfied;

1) The program in the population with the maximum fitness/entropy is greater than or equal to the user defined target fitness value, or

2) The current generation number is equal to the user defined maximum number of generations.

If neither of these criteria are met, then the next stage is to prepare a modified population for the next generation. If one or both of these criteria are met, the GP will terminate. Regardless of which outcome happens, data is gathered for that particular run and is saved to a text file in the format of a CSV[5](Comma Separated Values) file. Each output and it's type are defined in figure 4 below;

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

Figure 4: Output data

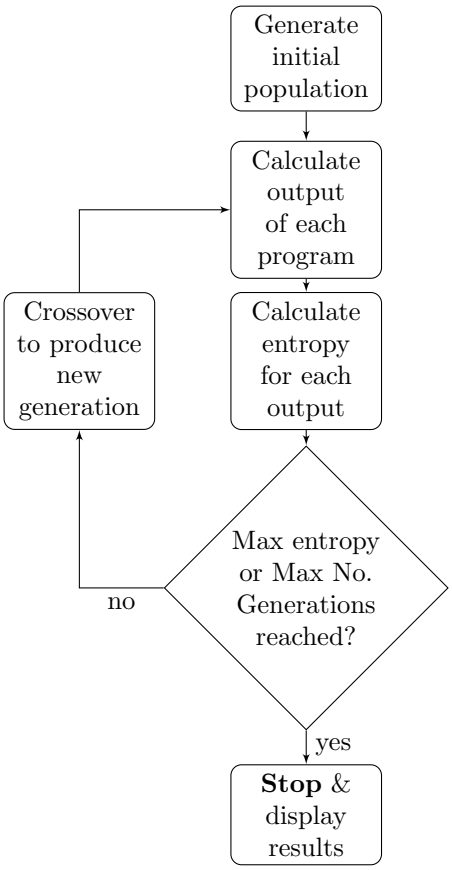
Data Type	Output description
Integer	Generation number
Double	Generation run time in seconds & hundredths of a second
Double	Average entropy/fitness for the population
Double	Total entropy/fitness of fittest candidate
Double	Entropy for subsequence size 1 of fittest candidate
⋮	⋮
Double	Entropy for subsequence 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 operations 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[6].

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 2 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 3: GP flowchart



In figure 2 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 3 on page 8. This process continues until one or both of the termination criteria are met as discussed above.

[6] 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 3 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 mactching 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 what state it is.

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 $E_{total}, E_1, E_2, ..., E_7$

3.2 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.

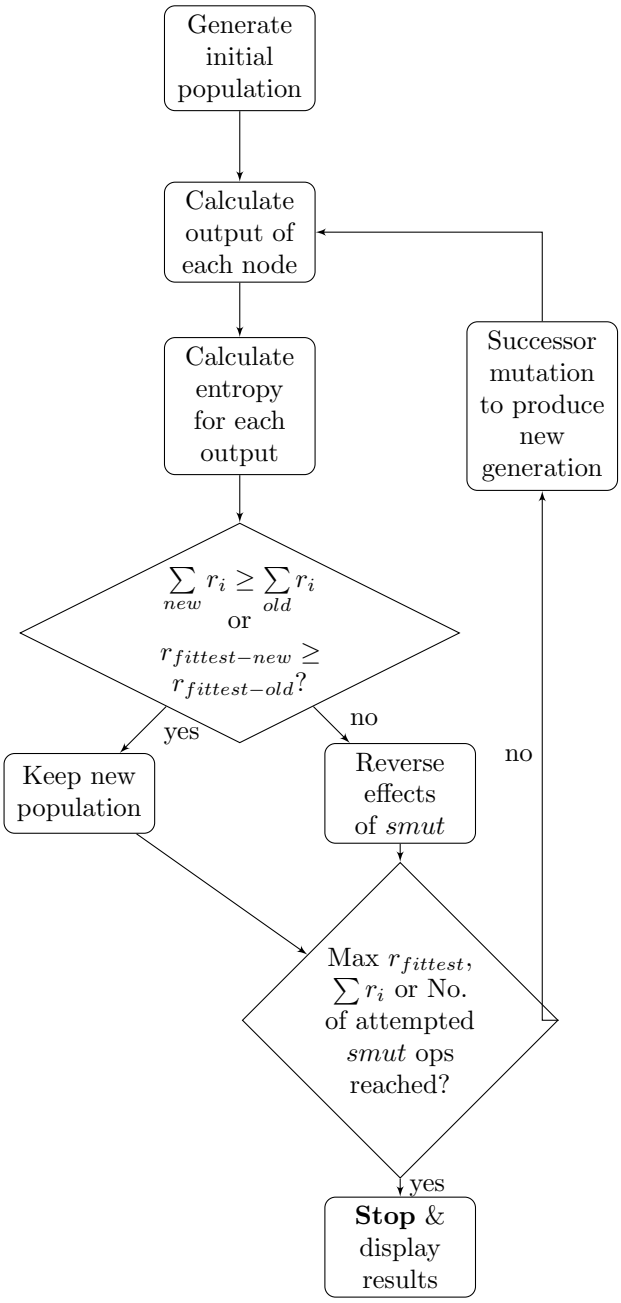
For this implementation, I will be adapting the fitness value r_i into a vector R_i [7]. The first element in the vector will be the total fitness E_{total} and the rest of the elements will be the scalar entropy values from E_1 to E_7.

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.

[7] 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 5 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 5: SNGP flowchart



The Inputs to the SNGP implementation are shown in figure 6 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.

Figure 6: 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.

Quick Evaluation Design

4 Implementation / Realisation

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. - GP implementation, go through algorithms to code.
- SNGP implementation, go through algorithms to code.
- C rand() and TRNG test implementation. - Describe changes; KMP matching algorithm, SNGP update list order realisation, added run data collection, integer to long for calculations, change of driving evolution in SNGP, terminating on smut to attempted smut.
- Problems encountered, SNGP producing incorrect entropy values for some trees,

5 Evaluation

5.1 Introduction

The first phase of evaluation is to compare the results I obtained from implementing Koza's method of producing randomisers with the use of Genetic Programming. 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 Congruent 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[8].

Both of these methods of producing random numbers are analysed using the same technique in the fitness function in our GP & SNGP implementations, by collecting 16834 random binary digits and calculating the Information Entropy for all possible binary subsequence's of sizes 1 to 7 bits. These two methods of generating random numbers in both deterministic and non-deterministic ways give us two solid competitors for the random number generators produced by our evolutionary methods. 50 runs of these two implementations were also collected in order to compare entropy and scalar entropy break down against our genetically bred RNGs.

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 5.2 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 5.6, 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.

5.2 Evaluation of GP Implementation

The first stage of implementation as discussed in the Realistion 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[9] 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[10] 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

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

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

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

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.

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 it 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 is 154.6 points, very close to the 153 points in Koza's fittest candidate[11] 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.

5.3 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 figure 7. 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].

Figure 7: GP and SNGP Standard Parameters

	GP	SNGP
Max Entropy	27.990 Bits	27.990 Bits
Max Generations / Mutation attempts	51	2500 [12]
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

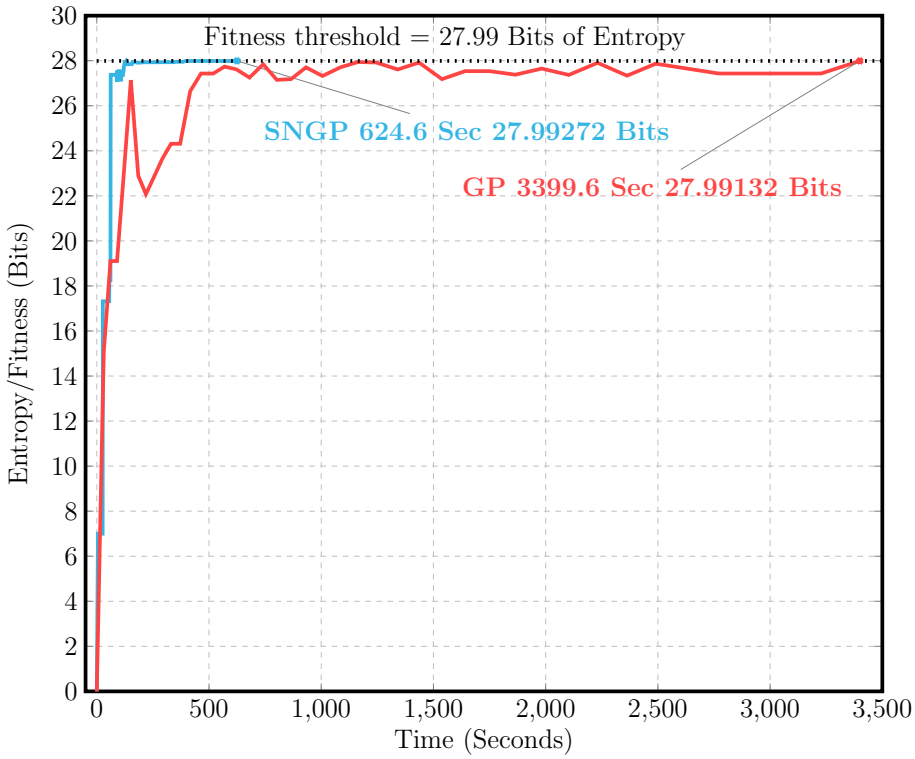
Now that we have defined the parameters, the data from 50 runs of the GP and SNGP implementations can be collected. The format of the data written by the runs can be found in section 3. The raw data written by the 50 runs can be found in the appendix, here this data shall be interpreted in order to conduct 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.

[11] Koza does not provide any average data or information for solution sizes.

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;

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 below 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

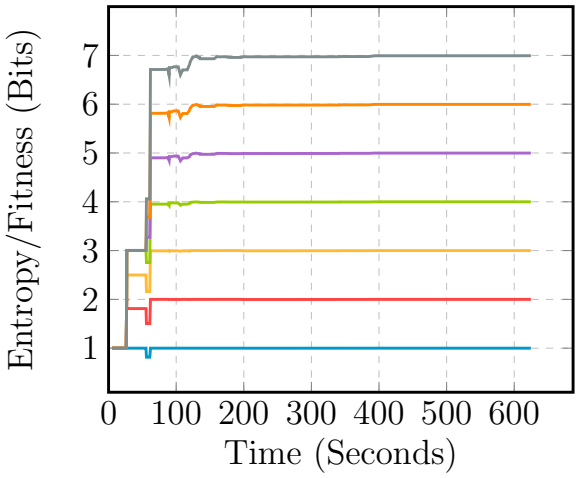
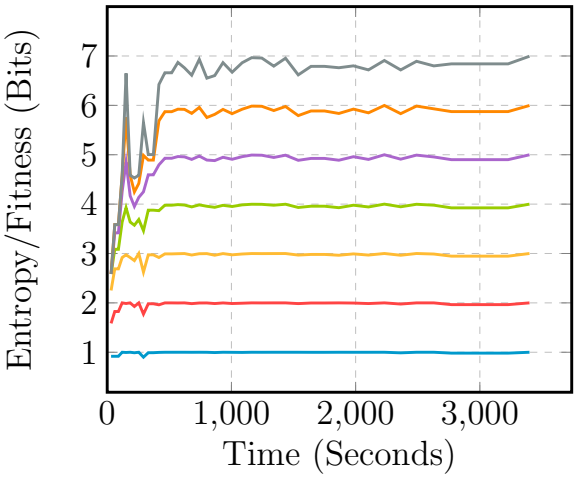
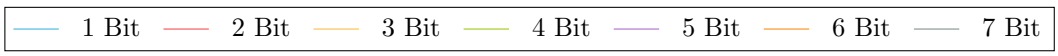


Figure 10: Scalar Entropies GP





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. After 50 runs of both implementations under standard parameters, the following average results were achieved shown in figure 11 below;

Figure 11: 50 Run Average: GP vs SNGP

	SNGP	GP
Avg Run Time	606.9s	4111.6s
Avg Sol Run Time	483.2s	2173.4s
Solution Rate	88%	40%
Avg Sol Size[13]	27.9	154.6
Avg Final Size[1]	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[14] 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. Figure 12 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 that on average 88% of solutions are found within the first 2500 *smut* attempts.

Figure 12: 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 $p_s = 0.88$, 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%[15] 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 **??** and **??** 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;

[14] Not including the initial generation

[15] Getting close but never quite 100% due to the limit of the function ($\lim_{x\to\infty} 1 - (1 - 0.88)^x = 1$)

Figure 13: GP surface plot

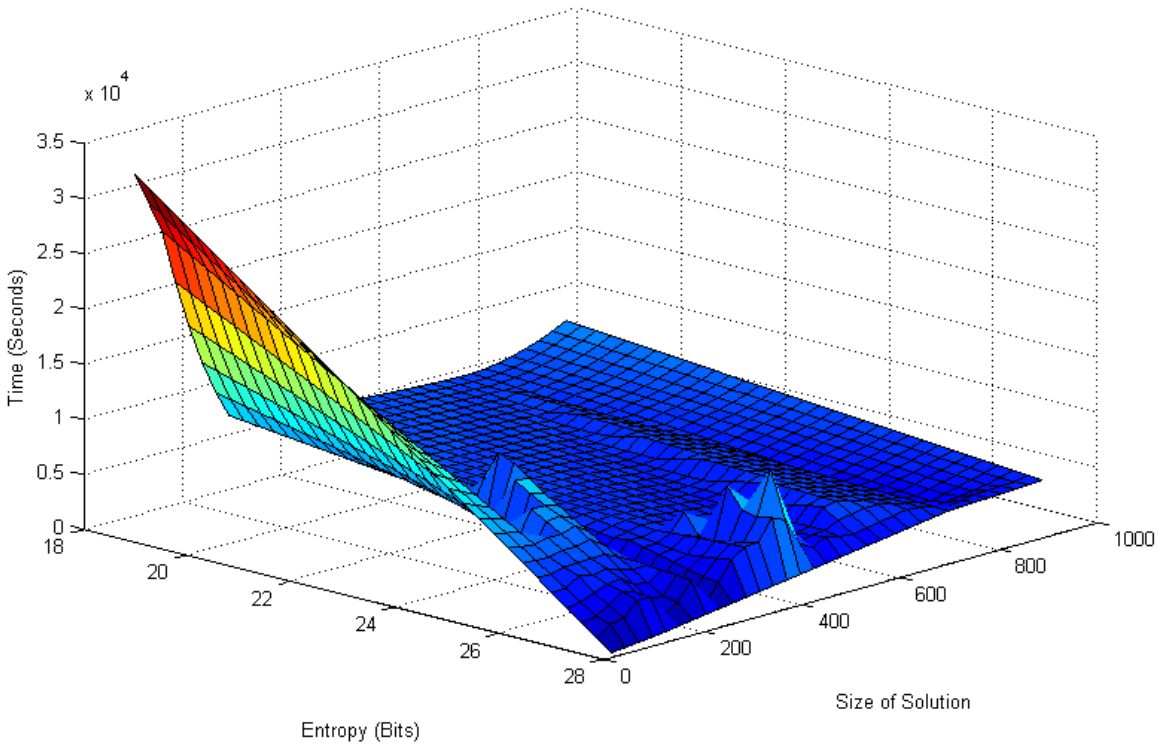
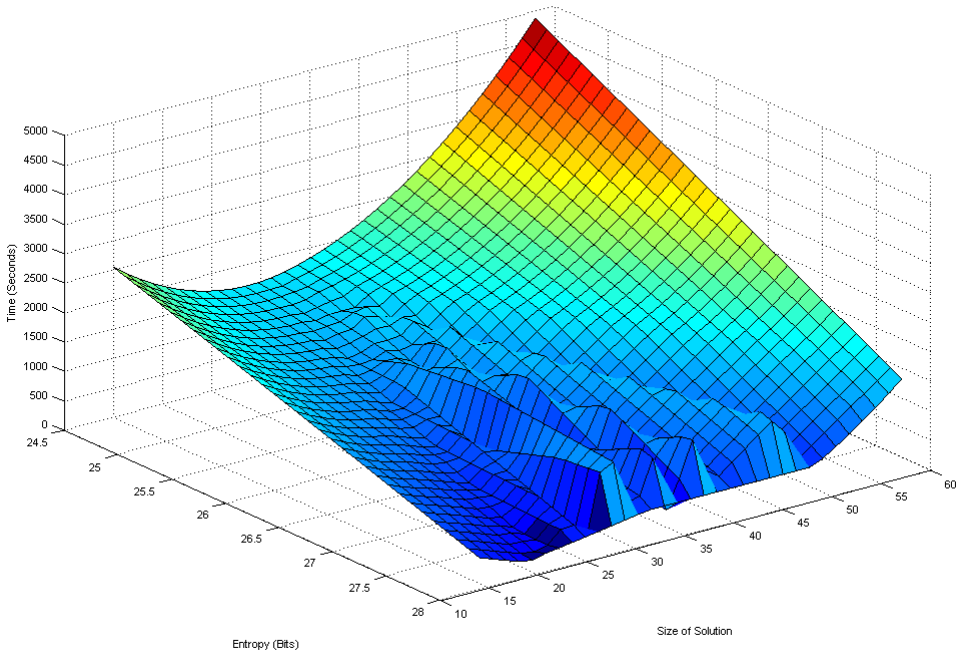


Figure 14: SNGP surface plot



We can conclude therefore 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 at the expense of extra computation time and it still have an average run time slightly faster than that of GP.

5.4 Parameter Altering Experimentation: GP vs SNGP Implementation

- Introduce experimentation with the run conditions of both GP and SNGP. Then show results and findings.

5.5 Genetic vs Conventional RNGs

- Introduce comparisons of Genetic vs Conventional means of producing RNGs and random numbers.

5.6 Real Implementation

Another way to assess the successfulness of this project is to see the real world application of the RNGs produced by evolutionary methods. Take the fittest and smallest solution produced by the GP implementation as discussed in section 5.2;

$$/ - 2 - 2 J \% J - 2 / / J 3 3$$

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 18, 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 19 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 15: C *rand*()

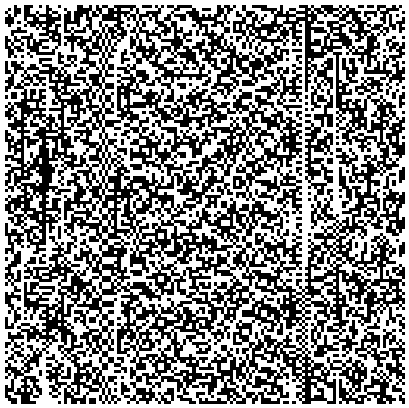


Figure 16: Random.org

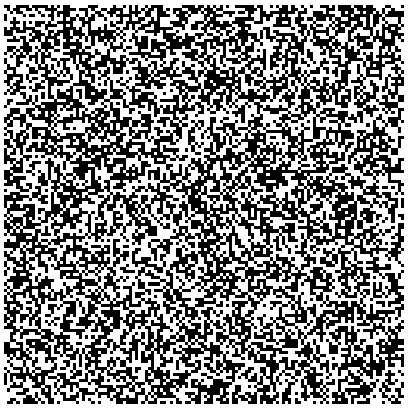


Figure 17: Genetic

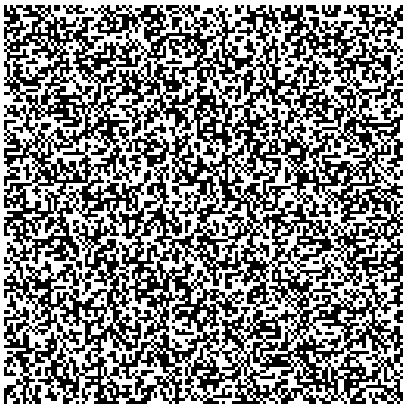


Figure 18: Logarithmic (base 10) Scale Run time of n calls of $rand(J) = \frac{(2-(2-J)}{J \mod (2-(\frac{J}{6}))}$ and C *rand*(), on the fixed min value of 0 and max value of 2000 for all runs

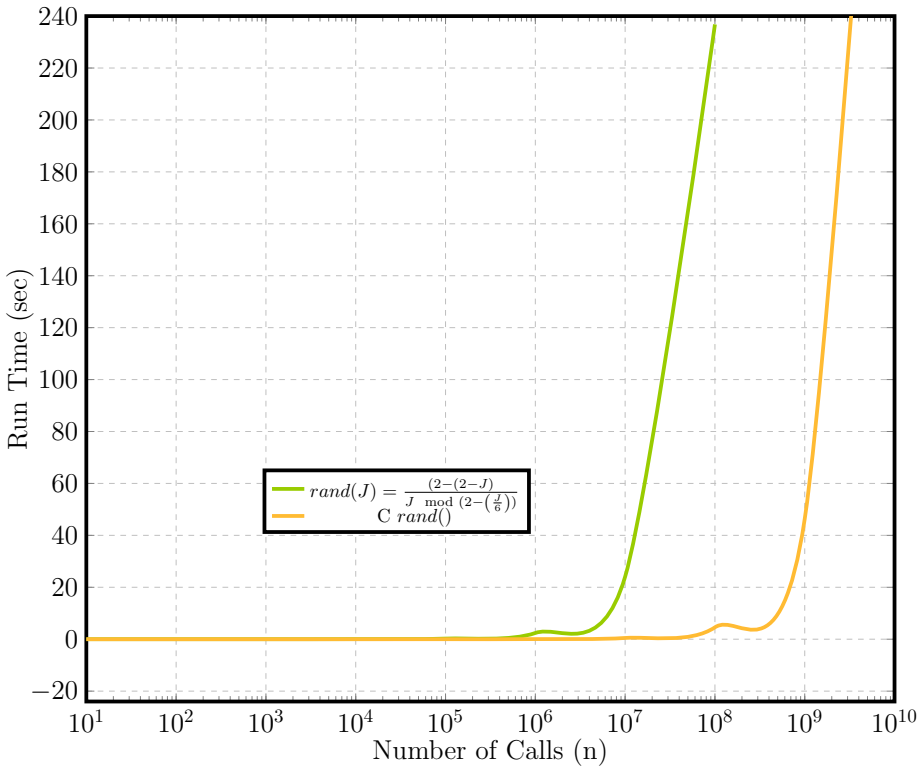
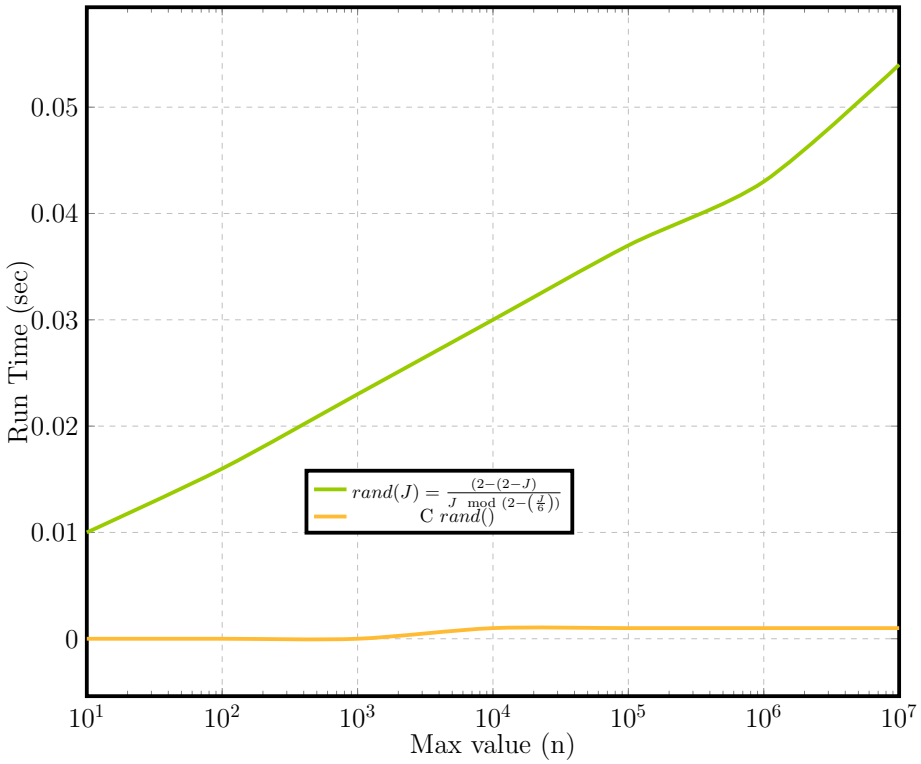


Figure 19: Effect of increasing max value of random number on run time, 1000 calls for each max value. Logarithmic (base 10) Scale Run time of $rand(J) = \frac{(2-(2-J)}{J \mod (2-\left(\frac{J}{6}\right))}$ and C $rand()$



5.7 Temporary Graphs

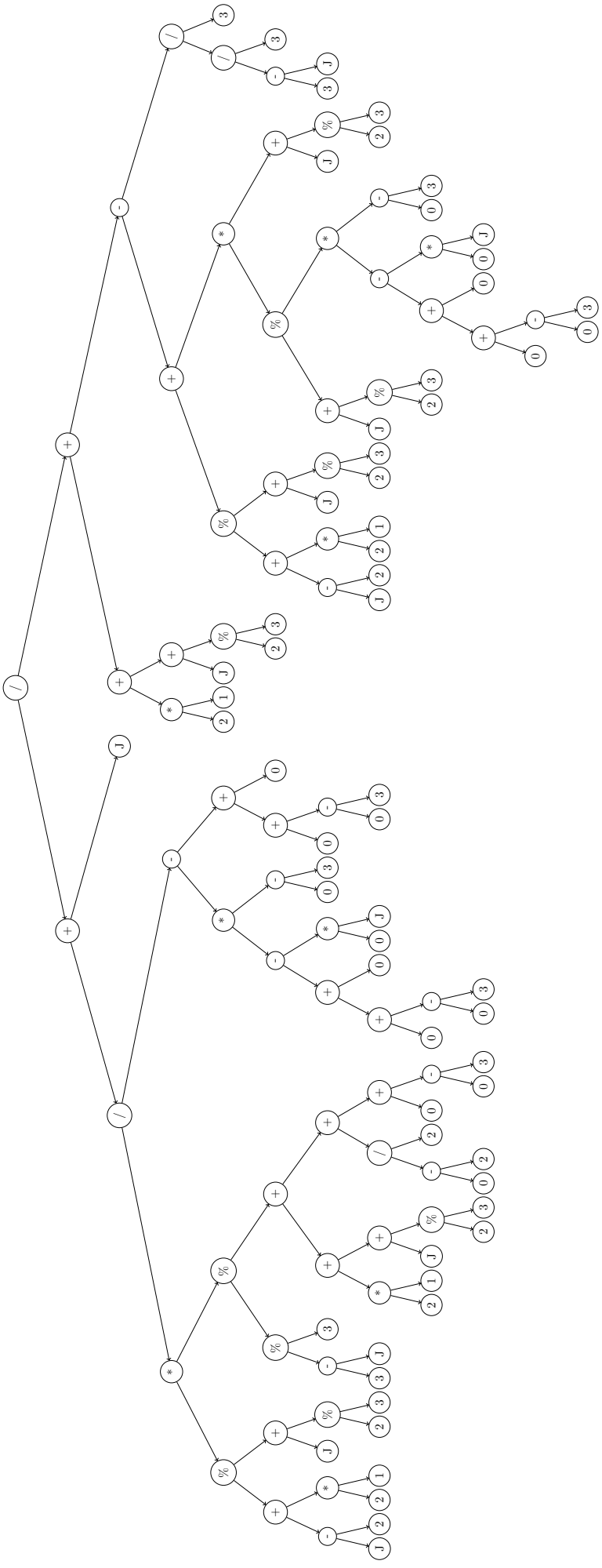
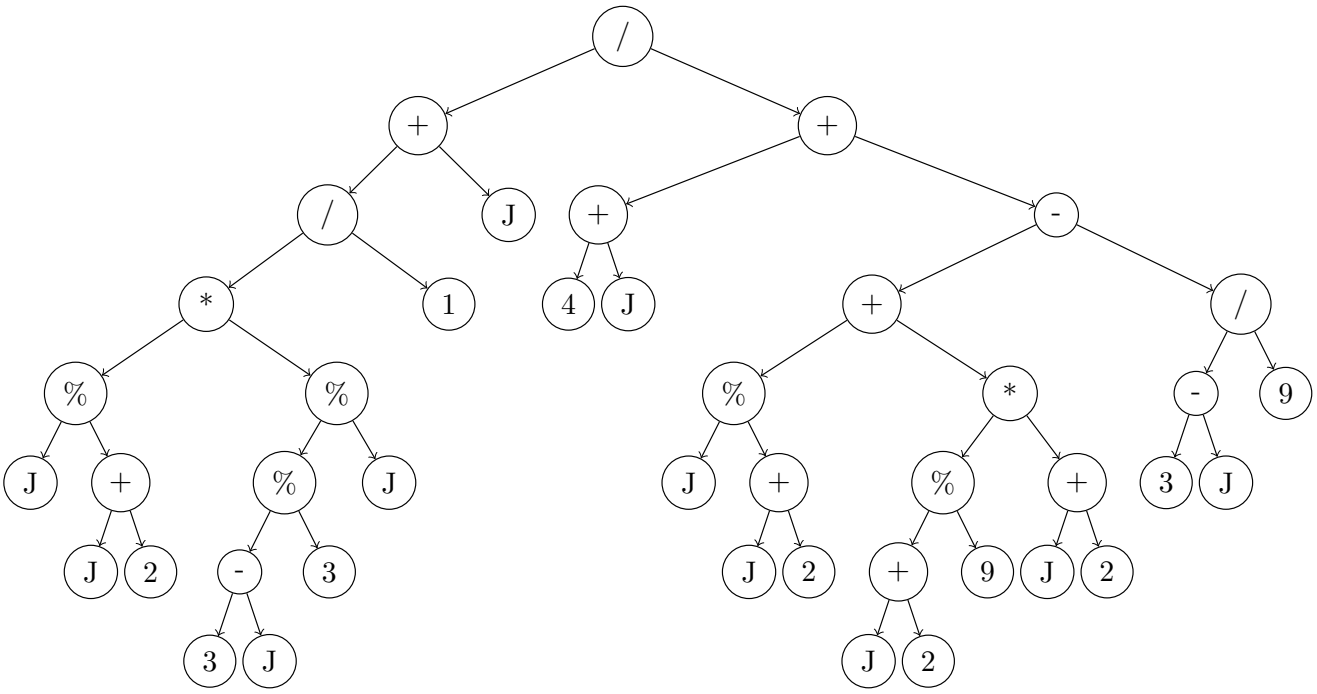


Figure 20: Fittest RNG Tree - 27.995729 Bits of Entropy

Figure 21: Simplified RNG Tree



The RNG tree in figure 20 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 this at compile time in order to reduce runtime in what's known as Constant Folding and Partial Redundancy Elimination. The tree in figure 20 can therefore be simplified into the tree in figure **??**. This simplified tree can be represented as an infix mathematical function $f(J)$ below;

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

Figure 22: Reduced Pop size of SNGP

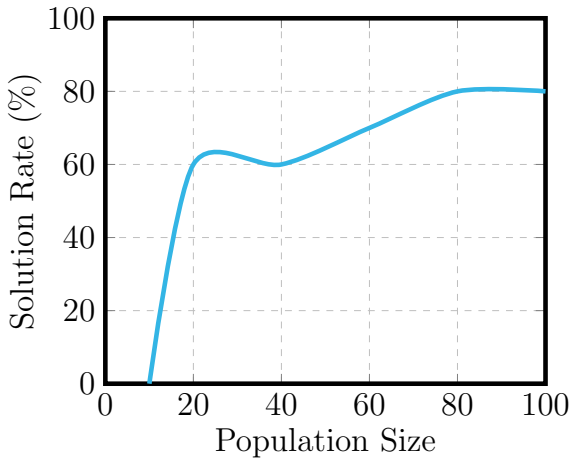
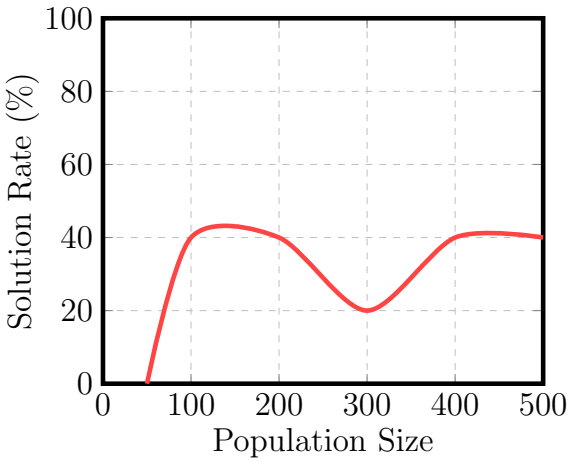


Figure 23: Reduced Pop size of GP



6 Learning Points

At least one page of summary of the key learning points in the project.

7 Professional Issues

At least one page of discussion of how your project related to the codes of practice and conduct issued by the British Computer Society.

8 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, *Single Node Genetic Programming on Problems with Side Effects*. University of Liverpool, 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, *A New, Node-Focused Model for Genetic Programming*. Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, Springer Verlag 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.

9 Appendices

Appendices are meant to contain detailed material, required for completeness, but which are too detailed to include in the main body of the text. Typically they should contain code listings, details of test data, screen shots of sample runs, a user guide, full design diagrams, instructions for unpacking and mounting any software included with the dissertation and similar material. A disc or CD-ROM containing the project archive material (including source codes, pictures used, and the dissertation document) should be submitted with the project report. Instructions on how to run the software from the disc/CD-ROM should be provided. If your system is available on-line, you should provide instructions of how to access the system via the internet.