Random Number Generation using Genetic Programming Design

Philip Leonard

Abstract

This is the design document 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 the summary of the project proposal, and primarily both the System and Evaluation Design. The last section covers the up to date plan for the project.

Keywords: Genetic Algorithm (GA), Genetic Programming/Program (GP), Pseudo Random Number Generator (PRNG), Random Number Generator (RNG), Entropy, Single Node Genetic Programming/Program (SNGP), Crossover, Mutation, Fitness proportionate reproduction (FPR).

Contents

1 Summary of Proposal

The aim of this project is to Evolve a Random Number Generator using Genetic Programming, and Single Node Genetic Programming methodologies and to compare both approaches. Main objectives of the project are as follows;

1) Implement Koza's idea of Evolving a Random Number Generator using Genetic Programming from his paper [1] as a C program.

2) Implement the same idea, but by using the Single Node Genetic Programming paradigm as described by Jackson in [9], again as a C program.

3) Finally compare both Genetic Programming methodologies, and then against common RNGs using evaluation methods designed in this document, in order to determine which approach to the problem is most effective.

There are some changes to the initial specification regarding the proposed user manual for the software, and what the programs will output. Both GP and SNGP programs will produce files including; the prefix expression of the fittest individual in the population, the time it took to complete a run of the program, and both maximum and average entropies for each generation in that run. The user manual will then be more extensive than originally anticipated, as it will provide guidance on how this information should be used for practical and experimental use. The mechanics of the programs will also be discussed in detail in order to allow users to adapt the software.

As discussed in the specification document, research has been the dominating factor in preparing for this project. I began researching Genetic Algorithms [3, p.1 - 24] and Genetic Programming [5, p.1 - 35] [6, p.73 - 191] topics. Since then I have researched more specifically into Genetic Programming, including a C implementations of Multiplexer Problem described in [7]. So far the focal point of my research has been around the paper by Koza [1], which is the inspiration for the implementation of this project using GP. Since creating the Specification document, the next key bit of research has been into SNGP, using [9] to formulate the design for the second objective.

2 Design

2.1 System Design

Both GP and SNGP approaches will be implemented in the C programming language. C will provide computational leverage over other programming languages whilst also providing effective ways of dealing with program structures and manipulating memory, making it a highly suitable candidate for the job. Both implementations will each require a single C program file, and will require no parallelism/threading because of GP and SNGPs sequential problem solving style. As a result of these characteristics, the best approach to the design of the project is to use a traditional methodology, describing it at a functional level. This includes a written design of the components, flow charts and the respective pseudocode of the key methods for both GP and SNGP solutions. Finally the evaluation design will encompass the test methods, conditions and comparison techniques used in order to evaluate and compare both approaches, and to pit them against other widely used RNGs.

2.1.1 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[1] from the output of the RNG on the i^{th} input. For example, given a RNG which outputs 12 on the 5^{th} iteration (input $i = 5$), the value of the 5^{th} digit in the binary sequence will be 0, as the LSB of 1100 ($12_{10} = 1100_2$) is 0.

As explained in [5, 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;

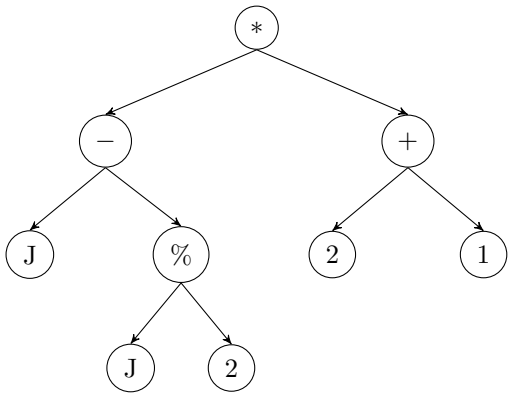
$$T = \{J,\ \Re\}$$

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 [5, 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". In the full method, trees are filled on all branches right up to the maximum initial depth. This is done by recursively choosing nodes from F to populate the tree until the depth is one less than the maximum. The remaining nodes are filled as randomly selected elements of T. The grow function on the other hand randomly selects an element from the set $F \cup T$ until all branches are terminated with an element of T (still limited by the maximum depth). If the selected element is a function then it has an arity of 2, so two more branches are recursively generated. Otherwise the selected element is a terminal which has an arity of 0, and therefore no further leaf nodes are generated. 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

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

half and half method, we can obtain half a population of large programs, and another half of programs with varying shapes and sizes.

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 techniques at our disposal. In this GP, we shall use the entropy of a given output to correspond to the fitness of that RNG. 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 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). For example take the binary sequence 1111, and take $h = 2$. Here $E_2 = 0$ as there are 4 possible binary subsequences of length 2; $\{00, 01, 10, 11\}$, and only 11 occurs in the sequence 1111. Therefore $1 \log_2 1 = 0$, and for the 0 probabilities of the other sub strings; $0 \log_2 0 = 0$. The negated sum of these values is 0, and therefore the sequence 1111 achieves the minimal entropy for sub sequences of length 2. Take another binary sequence 10011, for $h = 2$. This sequence obtains the maximum entropy for E_2, as the binary sub sequences of length 2; $\{00, 01, 10, 11\}$ occur with equal probabilities; $\{0.25, 0.25, 0.25, 0.25\}$ so the calculation of the entropy would be;

$$\begin{aligned} E_2 &= -(0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) \\ &= -(-0.5 + -0.5 + -0.5 + -0.5) \\ &= 2 \end{aligned}$$

Therefore 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, I 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;

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

In my GP, I shall be using sub sequences of lengths 1 to 7, so $N_{max} = 7$. This will give me the maximum entropy and therefore the maximum fitness of 28;

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

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[2](Comma Separated Values) file. Each output and it's type are defined in figure 4 below;

Figure 3: GP flowchart

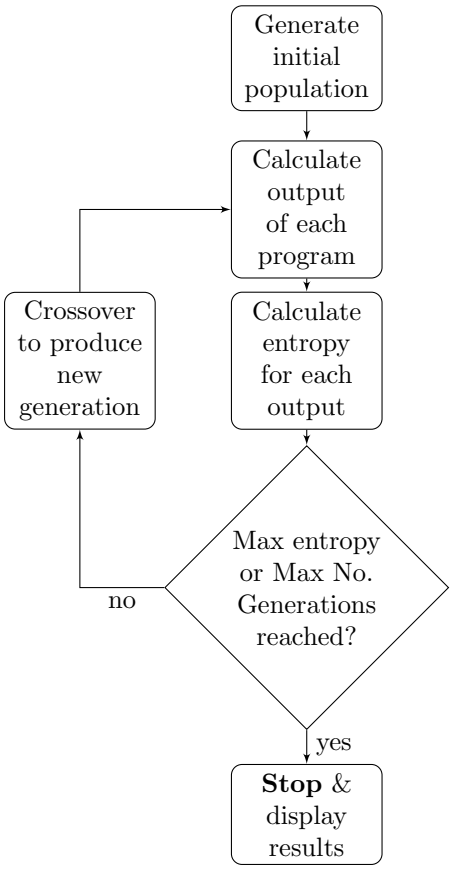


Figure 4: Output data

Data Type	Output description
Integer	Generation number
Double	Generation run time in seconds & hundredths of a second
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

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 example, after 3 generations the file could look like the following[3];

```
1, 12.51, 1.0, 1.9, 0.8, 0.0, 0.0, 0.0, 0.0, 3.7, FALSE, +J2
2, 15.23, 1.0, 2.0, 2.8, 2.1, 0.2, 0.0, 0.0, 8.3, FALSE *%J2-J1
3, 24.32, 1.0, 2.0, 3.0, 4.0, 2.9, 1.2, 0.0, 14.1, FALSE *-J%J2+21
```

Gathering and storing data in this way will aid in the evaluation of the GP against SNGP (where data will be gathered in a similar manner). Storing it in the CSV format will allow me to easily import the data into Microsoft Excel, MATLAB and LaTeX in order to conduct statistical analysis and construct graphical representations. Including this as a component of the software will also allow future users to conduct their own evaluation and analysis. The user manual will also contain information and instructions on how to manipulate this data in the software mentioned above. The files created will contain a time stamp in their name so that a user can conduct multiple runs without having to worry about the data from the last run being overwritten. This will aid evaluation later on.

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 [5, p.15-17]. Both operate by altering program trees in

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

[3]The purpose of the example data here is to display its format, and the entropy values are not correct for the corresponding prefix expressions

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[4].

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 3 is the probability divide of selection of internal to external nodes. In [6, 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 5 shows an example of subtree crossover for the two parent trees;

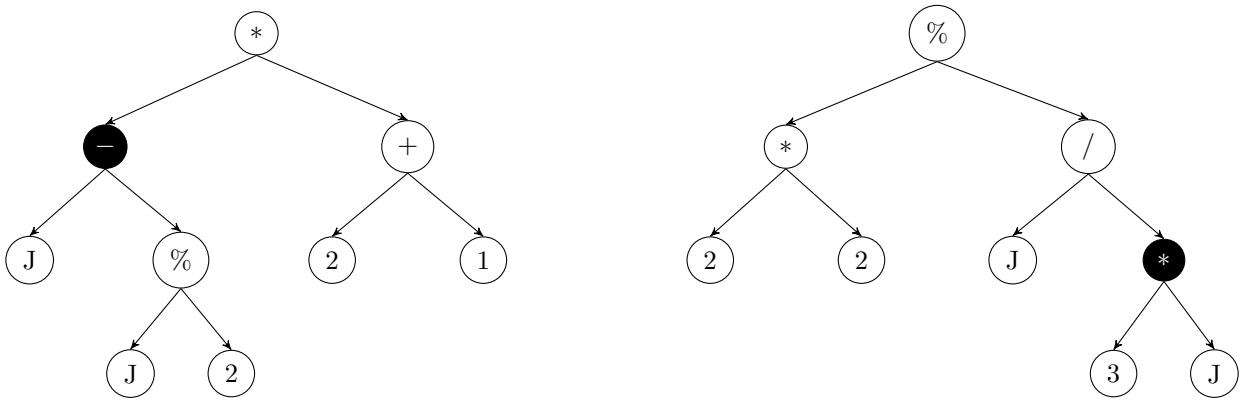
$$\{* \underbrace{-}_{\text{crossover point}} J \,\%\, J\ 2\ +\ 2\ 1\} \textbf{ and } \{\%\ *\ 2\ 2\ /\ J \underbrace{*}_{\text{crossover point}} 3\ J\}$$

which produce the offspring;

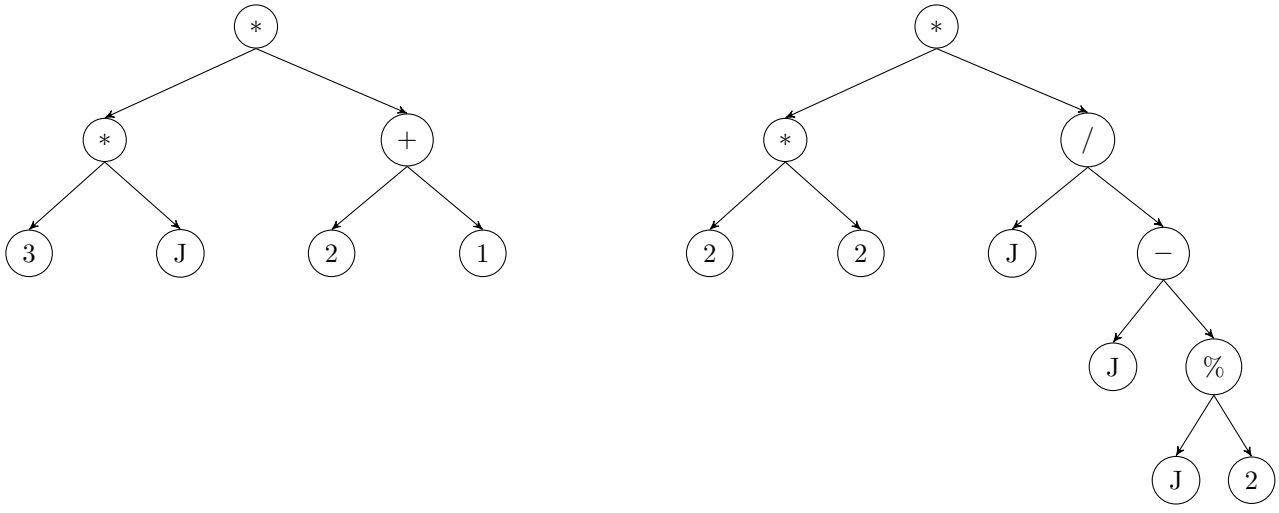
$$\{*\ *\ 3\ J\ +\ 2\ 1\} \textbf{ and } \{*\ *\ 2\ 2\ /\ J\ -\ J\,\%\,J\ 2\}$$

Figure 5: Example of subtree crossover

(a) Parents



(b) Offspring



In figure 2 on page 3, 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}

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

individual in the population (used for selecting both parents in FPR and one in crossover);

$$p_i = \left(\frac{f_i}{\sum_{x=1}^{|P|} f_x} \right) \qquad \text{where;} \qquad \begin{array}{l} p_i \text{ - selection probability of } i \\ f_i \text{ - fitness value of } i \\ f_x \text{ - fitness value of } x \\ |P| \text{ - population size} \end{array}$$

As Lapante explains in [4, p.63], we can imagine a roulette wheel where members of the population with higher fitnesses (i.e. probabilities) are given greater chunks. For FPR, both parents are selected at random relative to their probabilities (size of chunk on the roulette wheel). For crossover (in this case as described by Koza in [1, p.41]), one of the parents is selected in this manner and the other is selected where the probabilities of every member in the population is equal. A typical balance between the two methods is 90% crossover and 10% FPR. In this case, given a population of size 500, our new population is generated by conducting $500/2 * 0.9 = 225$ crossover operations and $500/2 * 0.1 = 25$ FPR operations.

If the proportion of FPR is too high, we are at risk of reducing genetic diversity, and creating two segregated sub populations similar to the biological effects of Allopatric or Peripatric Speciation. If this occurred in the GP, one population could be full of the fittest candidates, and the other contains the least fittest, and interbreeding could occur only rarely across the two populations.

Now that the GP has generated a new *evolved* population, it returns back to step 2 in figure 3 on page 5. This process continues until one or both of the termination criteria are met as discussed above.

Above is a literal design of the GP, covering all the major aspects of the operations in detail. Using the definitions above, a pseudocode can be created for the key methods in figure 3 on page 5 which bears a closer functional resemblance to the actual C implementation of the GP. I will use this pseudocode in Algorithms 1 to 6 in order to aid me in the implementation stage.

Algorithm 1 Generate-Initial-Population(P_s, M_{id})

Input: Population size P_s and Maximum initial depth M_{id}
Output: Population P of randomly generated programs of size P_s

```
1: i ← 1
2: for i to P_s do
3:     if i mod 2 == 0 then
4:         grow ← true
5:     else
6:         grow ← false
7:     end if
8:     partialTree ← ε
9:     T ← Create-Tree(partialTree, grow, M_id)
10:    add T to population P
11: end for
12: return P
```

Algorithm 2 Create-Tree($partialTree_i$, $grow$, d)

Input: Boolean value $grow$ to select grow or full function and Integer current depth d, and the tree with so far $partialTree_i$ with pointer i
Output: Recursively selects functions or terminals to produce a prefix expression

```
1: if depth == 1 then
2:     node ← random terminal
3: else
4:     if grow == true then
5:         node ← random function or terminal
6:         add node to partialTree_{i+1}
7:     else
8:         node ← random function or terminal
9:         partialTree_{i+1} ← node
10:        if node is a function then
11:            for i ← 1 to arity of function do
12:                partialTree ← Create-Tree(partialTree, grow, depth − 1)
13:            end for
14:        end if
15:    end if
16: end if
17: return partialTree
```

Algorithm 3 Calculate-Output(T, J)

Input: Tree prefix code T, Integer J
Output: Binary sequence O of size J

```
1: for i ← 0 to J do
2:     output ← Evaluate-Tree(T_0, i)
3:     O[i] ← output  mod 2
4: end for
5: return O
```

Algorithm 4 Evaluate-Tree(T_x, $currentJ$)

Input: Tree prefix code T_x with a pointer to the node x, Integer $currentJ$ which is the value of J in this instance
Output: Integer output of the subtree

```
1: node ← T_{x+1}
2: if node is + then
3:     return Evaluate-Tree(T_x, currentJ) + Evaluate-Tree(T_x, currentJ)
4: else if node is − then
5:     return Evaluate-Tree(T_x, currentJ) − Evaluate-Tree(T_x, currentJ)
6: else if node is ∗ then
7:     return Evaluate-Tree(T_x, currentJ) ∗ Evaluate-Tree(T_x, currentJ)
8: else if node is % then
9:     return Evaluate-Tree(T_x, currentJ) % Evaluate-Tree(T_x, currentJ)
10: else if node is / then
11:    return Evaluate-Tree(T_x, currentJ) / Evaluate-Tree(T_x, currentJ)
12: else if node is J then
13:    return currentJ
14: else
15:    return node                ▷ we know node must be ℜ, so return it's value
16: end if
```

Algorithm 5 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
        $occurrence \leftarrow 0$

        for $i \leftarrow 0$ to $|O|$ **do**
            if j_2 occurs at shift i in O **then** ▷ j_2 is j in base 2
                $occurrence \leftarrow occurrence + 1$
            end if
        end for
        $F \cup \{(j_2, occurrence)\}$
        $totalOcc \leftarrow totalOcc + occurrence$
    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 $E_{total}, E_1, E_2, ..., E_7$

Algorithm 6 Crossover-Operation(T^1, T^2, i_1, j_1 i_2, j_2)

Input: Two parent trees T^1 and T^2 with their respective crossover subtrees from i_1 to j_1 and from i_2 to j_2
Output: Two children trees T^1_{new} and T^2_{new}.

$T^1_{new} \leftarrow \emptyset$
$T^2_{new} \leftarrow \emptyset$

for $x \leftarrow 0$ to i_1 **do** ▷ copying up to crossover point in T^1
    $T^1_{new} \leftarrow T^1_{new} + T^1_x$ ▷ T^1_x denotes the character/node at position x in T^1
end for
for $y \leftarrow i_2$ to j_2 **do** ▷ copying crossover subtree in T^2 into T^1_{new}
    $T^1_{new} \leftarrow T^1_{new} + T^2_y$ ▷ T^2_y denotes the character/node at position y in T^2
end for
for $z \leftarrow j_1 + 1$ to $|T^1|$ **do** ▷ copying the rest of T^1 after the subtree into T^1_{new}
    $T^1_{new} \leftarrow T^1_{new} + T^1_z$ ▷ T^1_z denotes the character/node at position z in T^1
end for

for $x \leftarrow 0$ to i_2 **do** ▷ copying up to crossover point in T^2
    $T^2_{new} \leftarrow T^2_{new} + T^2_x$ ▷ T^2_x denotes the character/node at position x in T^2
end for
for $y \leftarrow i_1$ to j_1 **do** ▷ copying crossover subtree in T^1 into T^2_{new}
    $T^2_{new} \leftarrow T^2_{new} + T^1_y$ ▷ T^1_y denotes the character/node at position y in T^1
end for
for $z \leftarrow j_2 + 1$ to $|T^2|$ **do** ▷ copying the rest of T^2 after the subtree into T^2_{new}
    $T^2_{new} \leftarrow T^2_{new} + T^2_z$ ▷ T^2_z denotes the character/node at position z in T^2
end for

return T^1_{new}, T^2_{new}

Algorithm 7 Output-Data(g_n, f_{max}, E_{av}, E_1, E_2, E_3, E_4, E_5, E_6, E_7, T, F, $found$)

Input: g_n generation number, E_{total}, E_1, ..., E_7 total and scalar entropies of fittest candidate, E_{av} average entropy across all candidates, T prefix expression of fittest candidate, the file F and a boolean value $found$ whether the solution has been found in that run
Output: Write data to file, if successful then return 1, if not then return -1

$line \leftarrow g_n$, E_{total}, E_{av}, E_1, E_2, E_3, E_4, E_5, E_6, E_7, T, $found$ \n
open(F)
if write($line$, F) == true **then**
    close(F)
    return 1
else
    close(F)
    return -1
end if

2.1.2 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 [9] by Jackson.

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 [9, 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;

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

where;

$u_i \in \{T \cup F\}$ - 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

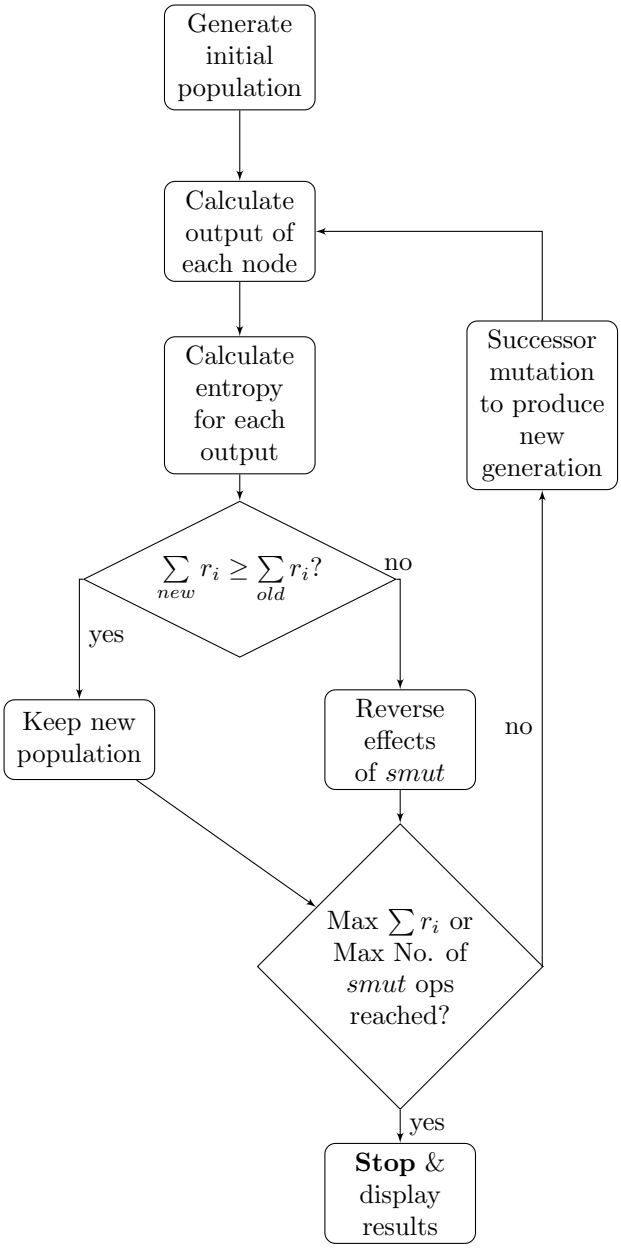
This tuple will be adapted into a *struct* in C like the GP program described above.

For this implementation, I will be adapting the fitness value r_i into a vector R_i [5]. 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.

Figure 6 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 6: SNGP flowchart



The Inputs to the SNGP implementation are shown in figure 7 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 smut operations. The terminating fitness value is taken from the sum of all fitness values in all the nodes of the population. 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 [9, p.52], this is what gives rise to SNGPs efficiency.

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

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 subsequences 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.

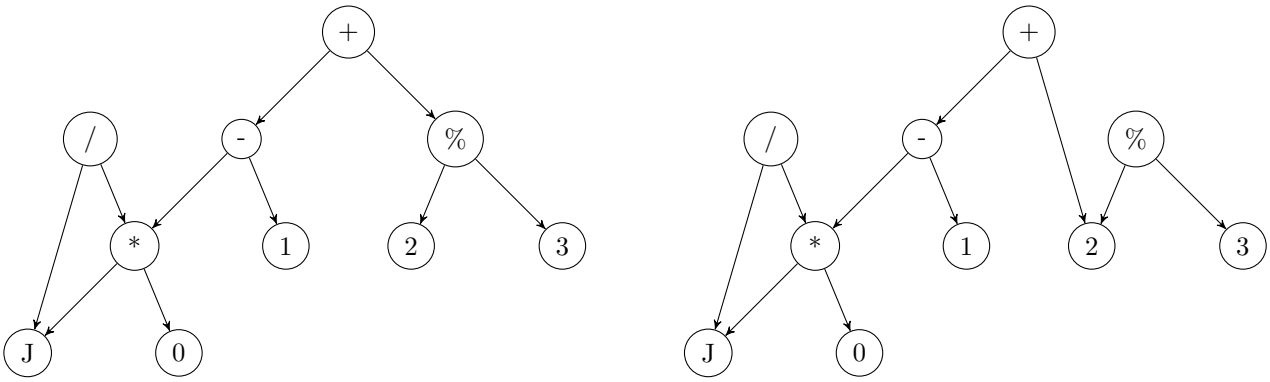
In SNGP, evolution is driven by hill-climbing. This is that the fitness is taken as the sum across the whole population; $\sum r_i$. After evolution this sum is calculated again, and if it is greater (higher entropy) than the last, then this evolved population replaces the previous population, otherwise the old population is kept and the process is repeated. This means that entropy/fitness can only climb higher. The total fitness $\sum r_i$ will be initially set to -1 so that the initial randomly generated population will always achieve a higher total fitness.

Figure 7: Input parameters

Data Type	Input description
Integer	Length of a run (number of *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). As we can see in figure 8, a member in the successor set for the root node "+", is mutated from "%" to "2";

Figure 8: Successor Mutation



In the example above, before mutation the population represents the prefix expressions (i.e. RNG trees);

$$\{J\},\ \{0\},\ \{1\},\ \{2\},\ \{3\},\ \{*\ J\ 0\},\ \{/\ J\ *\ J\ 0\},\ \{-\ *\ J\ 0\ 1\},\ \{\%\ 2\ 3\},\ \{+\ -\ *\ J\ 0\ 1\ \%\ 2\ 3\}.$$

After the *smut* operation it represents the following prefix expressions;

$$\{J\},\ \{0\},\ \{1\},\ \{2\},\ \{3\},\ \{*\ J\ 0\},\ \{/\ J\ *\ J\ 0\},\ \{-\ *\ J\ 0\ 1\},\ \{\%\ 2\ 3\},\ \{+\ -\ *\ J\ 0\ 1\ 2\}$$

During *smut*, not all of the program nodes are effected, and therefore not all of them need to be reevaluated for their outputs and fitnesses. As Jackson describes in [2, p.53] SNGP retains it's efficient nature by creating an update list, where only nodes effected by smut are added. Nodes in the list are reevaluated 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. The rest of the outputs will be the same as shown in figure 9 below;

Figure 9: Output data

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

The fittest candidate (i.e. best RNG tree) in a SNGP population is the tree where the root node has the maximum R_i across the whole population. This prefix expression is what is used to output to the file above in figure 9

Above is a literal description of the design for the SNGP implementation. A pseudocode for the key methods in the implementation can be created from the definitions above. Algorithms 4, 5 and 7 in section 2.1 are synonymous with the GP and the SNGP implementations. What differs where and when these algorithms are called, and how their outputs are stored. Algorithms 8, 9 and 10 below cover the altered and extra main methods in the SNGP implementation.

Algorithm 8 Generate-Initial-SNGP-Population(M_s, T, F, J)

Input: Population size M_s, set T of terminals and set F of functions, Integer J
Output: Population M of randomly generated programs of size M_s

```
1: i ← 1
2: for i to M_s do
3:     if T ≠ ∅ then
4:         node =random x ∈ T
5:         M ∪ node
6:         T = T − node
7:     else
8:         node =random x ∈ F
9:         j ← 0
10:        for j to node.arity do
11:            p ←random y ∈ M
12:            P_p ∪ node
13:            S_node ∪ p
14:        end for
15:
16:        T ← Get-Tree(node, ε)
17:        for i ← 0 to J do
18:            O_node[i] ← Evaluate-Tree(T_0, i)
19:            BinSeq[i] ← O_node[i]  mod 2
20:        end for
21:        R_node ← Fitness-Function(BinSeq)
22:        M ∪ node
23:    end if
24: end for
25: return M
```

Algorithm 9 Get-Tree($root$, T)

Input: Root node of tree $root$, existing tree prefix expression T
Output: Modified tree prefix expression T

1: $T = T + root$
2: **if** $S_{root} \neq \emptyset$ **then**
3: $\quad left \leftarrow$ left node of S_{root} for $root \in M$
4: $\quad T \leftarrow$ Get-Tree($left$, T)
5: $\quad right \leftarrow$ right node of S_{root} in $root \in M$
6: $\quad T \leftarrow$ Get-Tree($right$, T)
7: **end if**
8: **return** T

Algorithm 10 Successor-Mutation(m_i)

Input: Randomly selected member of the population, where $m_i \notin T$
Output: m_i with modified successor set

1: **repeat**
2: $\quad newSuc \leftarrow$ random $x \in M$
3: **until** $newSuc.depth > m_i.depth$
4: $S_i - random\ x \in S_i$
5: $S_i \cup newSuc$
6: **return** m_i

2.2 System Evaluation Design

Due to the thoroughness of data collection for both GP and SNGP in the designs above, there exists a variety of methods at my disposal in order to evaluate the project. Referring to the third point of the project objectives in section 1 of this document, we can see that the success of the software will be determined on two main evaluation premises. Primarily we shall design an evaluation, comparing the GP implementation against the SNGP implementation, which will use the data gathered across many runs of both. Secondly I shall design a comparison of the fittest GP and SNGP generated RNG trees against a widely used PRNG and also against a TRNG, again determining their randomness using a fitness function, calculating the entropy of a random binary sequence.

2.2.1 GP vs SNGP

In order to compare the GP and SNGP approaches, we must run the same input cases on both to obtain a set of outputs. To create these input cases we must only select the inputs which are synonymous between the two approaches and then leave the rest as a constant throughout all of the cases. Figure 10 below shows the matching input parameters between both implementations;

Figure 10: Matching parameters

Data Type	Input description
Integer	Maximum number of generations / Length of a run (number of *smut* operations)
Integer	Population size; number of trees (GP) or number of nodes (SNGP)
Double	Target fitness value (GP), Target fitness value for sum of all node targets (SNGP)

The remaining input parameters are only in the GP implementation. Figure 11 below shows the leftover parameters and their values[6] across all test cases for the GP implementation;

Figure 11: Remaining constant parameters for GP

Data Type	Input description	Values
Integer	Maximum initial program depth	4
Integer	Maximum crossover program depth	15
Double	Fitness proportionate reproduction probability	0.1
Double	Crossover operation probability	0.9
Double	Internal node crossover probability	0.9
Double	External node crossover probability	0.1

For figure 10 we must define a set of test cases concerning the values of these parameters. For each input I want to test it for a regular case and for an intensive case. For Maximum Generations/*smut* operations I want to test for a regular limit of 51 to a larger limit of 101. For Population size, I want to test it for a regular limit of 500 to a larger size of 1000. For the Target fitness, I want to test it for a regular entropy of 27.990 as described by Koza in [1, p.41], and a perfect fitness score of 28 (for SNGP this input value will be multiplied by that test cases population size). All combinations of these inputs give us p (possible values of each input) and k (the number of inputs), $p^k = 2^3 = 8$ test cases shown in the matrix below;

$$\text{Test Matrix} = \begin{array}{c} \\ \text{Max Gen/}smut \\ \text{Pop size} \\ \text{Targ Fit} \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 51 & 51 & 101 & 101 & 101 & 101 & 51 & 51 \\ 500 & 1000 & 500 & 1000 & 500 & 1000 & 500 & 1000 \\ 27.990 & 28.0 & 27.990 & 28.0 & 28.0 & 27.990 & 28.0 & 27.990 \end{pmatrix} \end{array}$$

Now that we have created test cases, both GP and SNGP will be run on these test cases which will generate time stamped files containing there respective outputs. Each test case can be run a number of times in order to produce outputs that we can average.

The fitness data and various properties of the program runs (i.e. average run time) can be used to pit the SNGP and GP against each other to determine which method performs best. For my evaluation I shall be comparing SNGP and GP using the following criteria;

1) Average entropy fitness of fittest member over 100 runs

2) Average solution rate over 100 runs

[6] These values are the ones which are defined by Koza in [1, p.41]

3) Average minimum and maximum solution size over 100 runs

3) Average run time over 100 runs

4) Generation run time over one run

5) Average solution size over one run

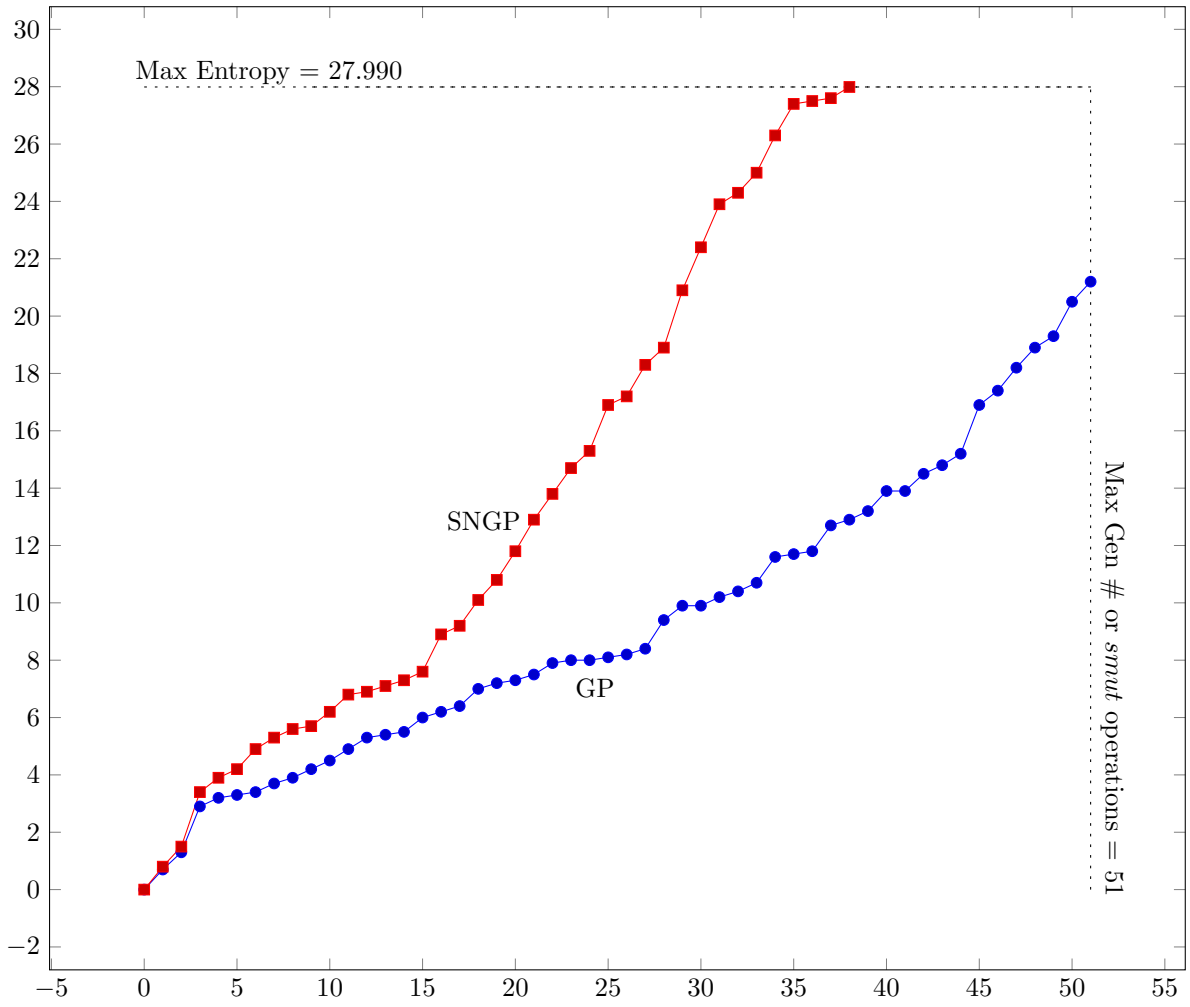
6) Entropy climb over one run

7) Scalar entropies over one run

There are also a variety of ways in which these comparisons can be represented in both visual and descriptive ways.

For my evaluation, I shall mainly be using charts to compare the statistics for both GP and SNGP outputs, and discussing their implications in terms of which method is best suited for their application. An example graph can be seen below in figure 12, comparing both SNGP and GP methods on a single run for the first test case in the matrix above;

Figure 12: Example evaluation graph



After comparison of both approaches, the expected result is that SNGP will outperform GP in most if not all evaluation criteria. The reason for this is that comparisons of the two methods conducted by Jackson in [2, p.55] display SNGP superiority over GP. As the fitness function and tree evaluation are computationally intensive (and a possible bottleneck for both approaches), SNGPs use of dynamic programming for both calculations should see it's run time substantially reduced in comparison to the GP implementation.

2.2.2 Other Random Number Generators

After methods for generating RNGs using GP have been evaluated, the actual RNGs produced should be compared to a widely used Pseudo Random Number Generator and a True Random Number Generator.

In order to evaluate these other RNGs, the same fitness function will be used so as to determine their randomness in terms of entropy. Therefore evaluation criteria will strictly be concerned with entropy as all the other criteria as described above are concerned with the GP functions.

2.2.3 C Rand function

The rand() function used in the C programming relies on the use of a PRNG. In order to calculate it's entropy, I shall write a small C program which call this function $i = 0$ to J times, each time finding the LSB of it's output and again populating a binary output vector. This vector can then be assessed for it's entropy using the same fitness function as in Algorithm 5 in section 2.1.1. This process will be repeated a number of times to obtain an average total entropy value and scalar entopies, which can be can then be compared to the fittest RNG produced by the GP/SNGP method.

2.2.4 True Random Number Generator comparison

As described in the specification document, a TRNG is a random number generator which employs a piece of hardware to generate a random number. As these pieces of hardware can be expensive and hard to come by I shall be making use of the free TRNG at http://www.random.org/. Random binary sequences at random.org are produced using a piece of hardware which measures atmospheric noise. This service allows users to make use of a free 1,000,000 bit quota per IP address (with a 200,000 bit top up each day to refill the 1,000,000 quota). These sequences are available over the Hyper Text Transfer Protocol (HTTP), therefore I shall be writing another simple C program to request a binary sequence of length J over HTTP. The program will then extract the data from the packet and determine the sequences fitness using the same fitness function in Algorithm 5 to determine it's entropy and scalar entropies. I shall repeat this process a number of times (remembering that $\lfloor \frac{1{,}000{,}000}{J} \rfloor = 61$ tests) to obtain an average entropy produced by the TRNG. Using this information I can then compare it with the entropy of the fittest RNG produced by the GP/SNGP method.

Using all of these evaluation methods will give a clear conclusion to;

a) Which method of Generating Random Number Generators using Genetic Programming is most successful and;

b) Whether the best RNGs produced by these approaches are a viable option compared to other (non genetically evolved) RNGs

3 Plan Progress

The progress of the project is going well, no rearrangements or readjustments need to be made to the Gantt chart, just the movement of the current date in order to show what aspects of the project have been completed. These are namely, the Specification, Research and the Design, and so reaching the milestone "Implementation documentation complete".

The next stage to the project after the presentation is implementation, specifically the implementation of the GP approach. I shall be using the algorithms described in section 2.1 to write the program in C.

Please See figure 13 on page 18 for the up to date Gantt Chart

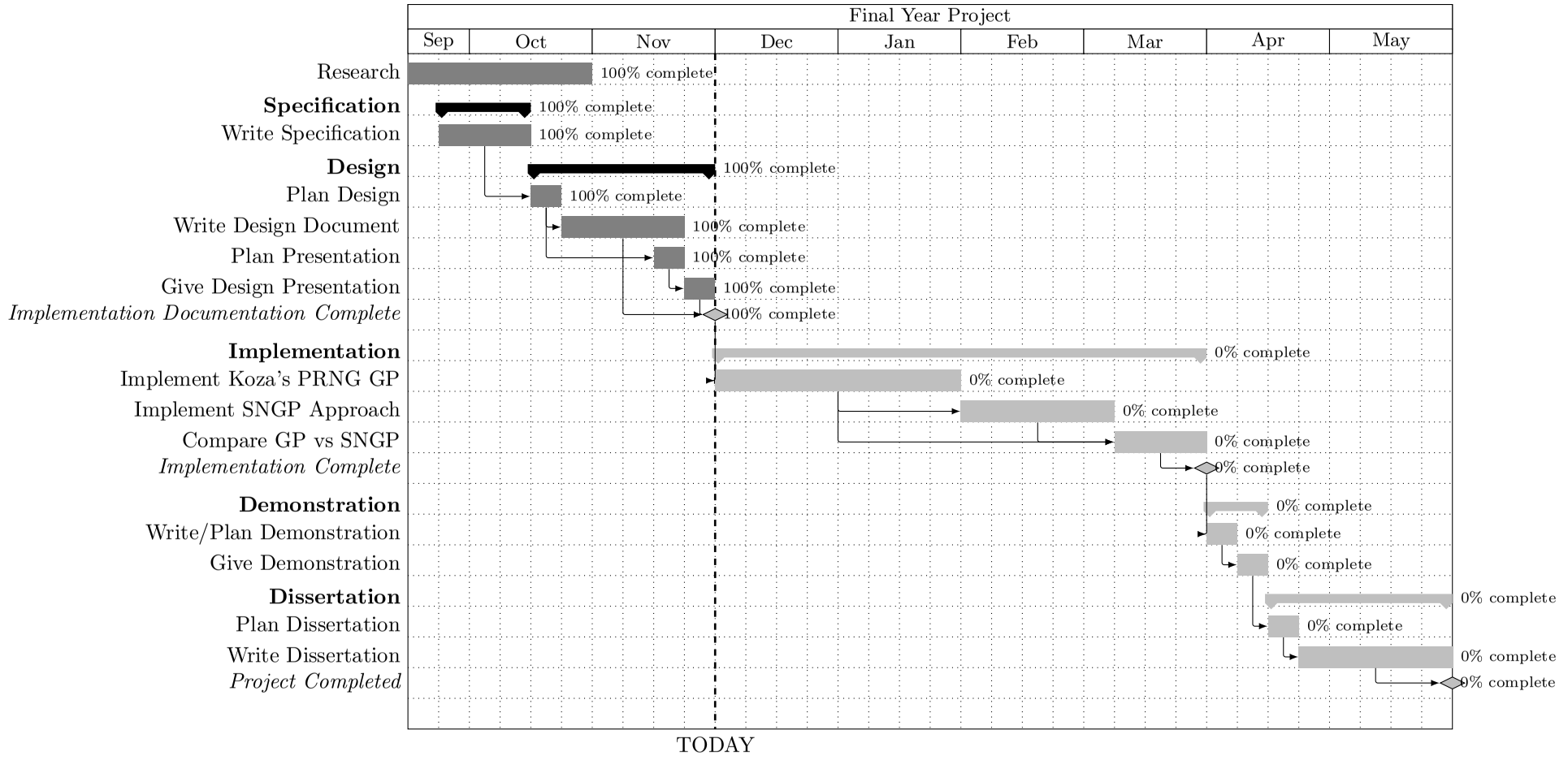


Figure 13: Gantt Chart - Work plan

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

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

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

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

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

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

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

[10] 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.

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