Random Number Generation using Genetic Programming Interim Report

Philip Leonard

Abstract

This is the Interim 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).

Keywords: Genetic Programming/Program (GP), Pseudo Random Number Generator (PRNG), Random Number Generator (RNG), Entropy, Single Node Genetic Programming/Program (SNGP).

1 Completed Work

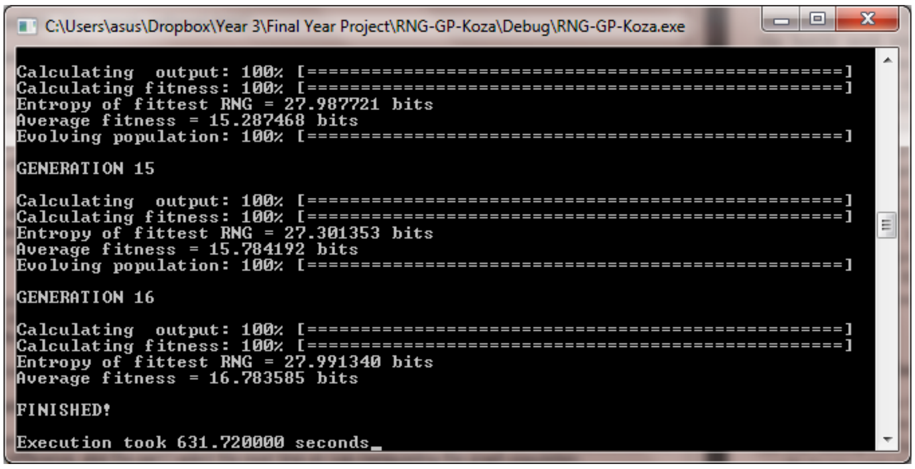
1.1 GP Implementation

I first completed the Genetic Programming implementation around November 2013. Upon doing test runs I realised that it was taking much longer than I had first anticipated. By using the clock function in C (which calculates CPU time), I calculated the running times of the main procedures and narrowed down the bottle neck, which happened to be the fitness function for finding the occurrences of binary subsequences. I substituted the brute force matching algorithm with the KMP (Knuth Morris Pratt) algorithm, an efficient finite automaton based pattern matching algorithm. By doing this I was able to reduce the running time drastically, however this still remains the bottle neck (highest order operation) in the program.

While the running time of the GP implementation remains slow in my eyes, the results of the runs are promising. To begin with I was surprised that solutions needed more generations to find a solution than I originally anticipated. But upon considering that the search space for this problem is relatively large, and that Koza defines a solution as a RNG with extremely high entropy I came to accept this.

In [1, p.6], Koza says that he finds a solution in 14 generations. Some runs of my implementation may only take so long (as seen in the example run & fig 1), but they can take around 30 to 60 generations to find one. As Koza says in [1, p.6], entropy slowly improves in the region of 27.800 to 27.900 bits, and as this method isn't driven by hill climbing, the entropy value fluctuates.

RNG Tree (27.991340 bits of entropy)
+/+-/-/J3-**/-/J3-+J1/J23/+3+**//J233+J33



```
C:\Users\asus\Dropbox\Year 3\Final Year Project\RNG-GP-Koza\Debug\RNG-GP-Koza.exe

Calculating  output: 100% [==================================================]
Calculating fitness: 100% [==================================================]
Entropy of fittest RNG = 27.987721 bits
Average fitness = 15.287468 bits
Evolving population: 100% [==================================================]

GENERATION 15

Calculating  output: 100% [==================================================]
Calculating fitness: 100% [==================================================]
Entropy of fittest RNG = 27.301353 bits
Average fitness = 15.784192 bits
Evolving population: 100% [==================================================]

GENERATION 16

Calculating  output: 100% [==================================================]
Calculating fitness: 100% [==================================================]
Entropy of fittest RNG = 27.991340 bits
Average fitness = 16.783585 bits

FINISHED!

Execution took 631.720000 seconds_
```

Figure 1: GP implementation example run screenshot

The time that can be seen in fig 1 is measured in CPU time (the accumulative time the thread has a hold on the CPU), and not real time. Otherwise this would reflect the speed of the CPU and not necessarily the efficiency of the program. After being generally pleased with the outcomes of the GP implementation I moved onto the SNGP implementation.

1.2 SNGP Implementation

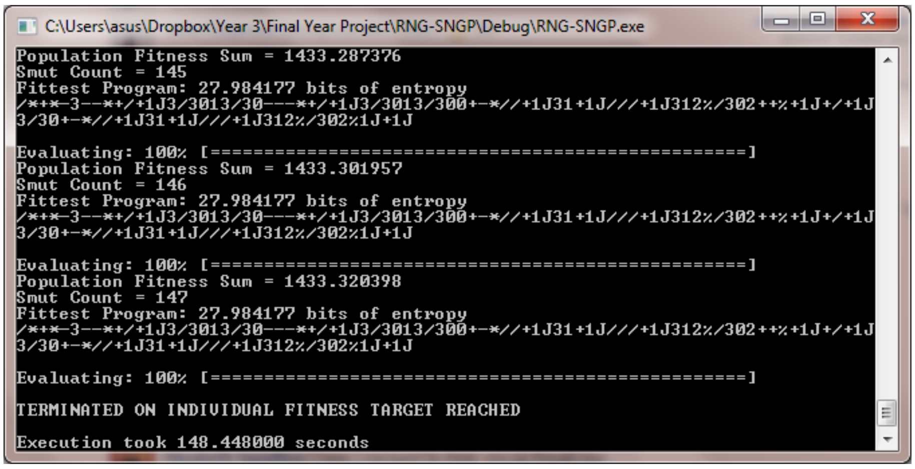
The next step was to implement the SNGP program. Having completed the GP implementation, I was able to quickly complete this due to there similarities. The main difference, and the part I spent the most time on was manipulating the graph population.

To begin with I implemented the program where I updated every node in the graph. Once I had this running correctly, I then went about implementing an update list method of determining the nodes that needed to be updated by recursively finding the parent(s) of the mutated node. Using this dynamic programming method, the run time of the program drastically improved.

From the runs that I have tested the program with it seems that SNGP is a faster method for finding high entropy RNGs than standard GP is. I was also surprised that this method was taking more hill climbing smut operations than I first anticipated, but I then realised that I was giving as an input an inordinately large fitness sum level for it to achieve (POP_SIZE * 27.90). I then changed the implementation slightly to include a stop condition of a single RNG fitness threshold, and lowered the fitness sum target (as having every tree in the graph being of such a high fitness was proving near impossible).

After doing this the SNGP approach began producing good results, and although still taking quite a number of smut operations, it was executing far quicker than the GP approach (as can be seen in 2 below).

RNG Tree (27.984177 bits of entropy)
/*+*-3–*+/+1J3/3013/30—*+/+1J3/3013/300+-*
//+1J31+1J///+1J312%/302++30+-*//+1J31+1J///+1J312%/302%1J+1J



```
C:\Users\asus\Dropbox\Year 3\Final Year Project\RNG-SNGP\Debug\RNG-SNGP.exe
Population Fitness Sum = 1433.287376
Smut Count = 145
Fittest Program: 27.984177 bits of entropy
/**-3--**/+1J3/3013/30---**/+1J3/3013/300+-*//+1J31+1J///+1J312%/302++%+1J+/+1J
3/30+-*//+1J31+1J///+1J312%/302%1J+1J

Evaluating: 100% [================================================]
Population Fitness Sum = 1433.301957
Smut Count = 146
Fittest Program: 27.984177 bits of entropy
/**-3--**/+1J3/3013/30---**/+1J3/3013/300+-*//+1J31+1J///+1J312%/302++%+1J+/+1J
3/30+-*//+1J31+1J///+1J312%/302%1J+1J

Evaluating: 100% [================================================]
Population Fitness Sum = 1433.320398
Smut Count = 147
Fittest Program: 27.984177 bits of entropy
/**-3--**/+1J3/3013/30---**/+1J3/3013/300+-*//+1J31+1J///+1J312%/302++%+1J+/+1J
3/30+-*//+1J31+1J///+1J312%/302%1J+1J

Evaluating: 100% [================================================]

TERMINATED ON INDIVIDUAL FITNESS TARGET REACHED

Execution took 148.448000 seconds
```

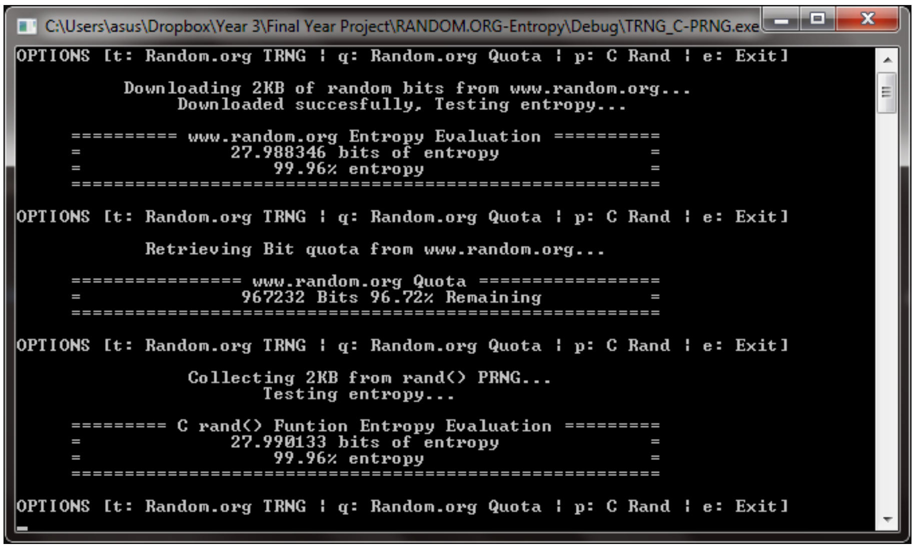
Figure 2: SNGP implementation example run screenshot

1.3 TRNG & C rand() Test program

The final part of the implementation phase was to implement the program to test both the Random.org TRNG and the C rand() function. Random.org supplies an API for which a sequence of Random bits produced by there instruments can be requested over the web. I implemented the program so that 16384 bits are requested over HTTP and then saved to a file on the users hard disk. This file is then read into a char* and the entropy is then tested in the exact same way as that of a RNG in the GP and SNGP approaches. The program then outputs the entropy data to the user which can then be collected.

The program also allows the user to check how many bits they have left in the quota (for the current ip address), out of 1,000,000 bits which Random.org provides for free.

The program can also test the C rand function. The program seeds the C rand function with the current system time and then requests 16384 bits from the rand function in order to populate a char*. The entropy is then tested in the exact same way and the results are shown to the user. See fig 3 for a example runs of both.



```
C:\Users\asus\Dropbox\Year 3\Final Year Project\RANDOM.ORG-Entropy\Debug\TRNG_C-PRNG.exe
OPTIONS [t: Random.org TRNG | q: Random.org Quota | p: C Rand | e: Exit]

          Downloading 2KB of random bits from www.random.org...
               Downloaded succesfully, Testing entropy...

     ========= www.random.org Entropy Evaluation =========
     =              27.988346 bits of entropy            =
     =                  99.96% entropy                   =
     =====================================================

OPTIONS [t: Random.org TRNG | q: Random.org Quota | p: C Rand | e: Exit]

            Retrieving Bit quota from www.random.org...

     =============== www.random.org Quota ================
     =               967232 Bits 96.72% Remaining        =
     =====================================================

OPTIONS [t: Random.org TRNG | q: Random.org Quota | p: C Rand | e: Exit]

                Collecting 2KB from rand() PRNG...
                       Testing entropy...

     ======== C rand() Funtion Entropy Evaluation ========
     =              27.990133 bits of entropy            =
     =                  99.96% entropy                   =
     =====================================================

OPTIONS [t: Random.org TRNG | q: Random.org Quota | p: C Rand | e: Exit]
_
```

Figure 3: TRNG & C rand() test program screen shot

2 Remaing Work

Th remaining work to be completed is as follows;

- Gather data from multiple runs of each method
- Compare all approaches using test criteria described in the design document
- Finally, give demonstration and write dissertation based on findings

By referring to the Gantt chart in fig 4, we can see that the project is about 1 month ahead of schedule. The next stage is to gather and compare the data from all approaches.

References

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

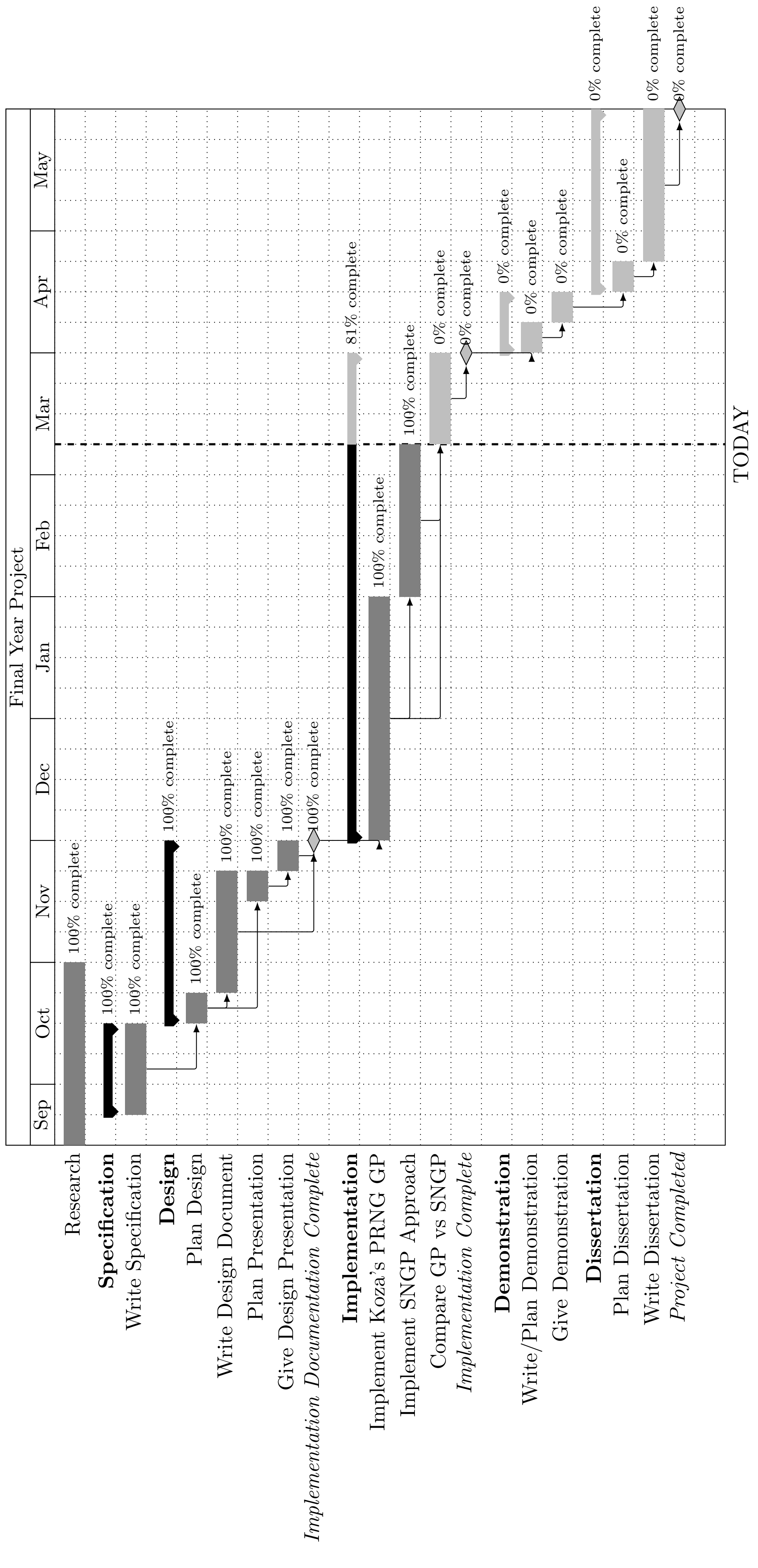


Figure 4: Gantt Chart - Work plan