Random Number Generation using Genetic Programming: Demonstration

Philip Leonard

University of Liverpool

Primary Supervisor: Dr. David Jackson
Secondary Supervisor: Professor Paul E. Dunne

sgpleona@student.liverpool.ac.uk

Presentation Overview

Genetic Programming - Struct

Here we can see how each tree is described as a struct in C.

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

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

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

Single Node Genetic Programming - Struct

Here we can see how each node is described as a struct in C.

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

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

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

Fitness Function - GP & SNGP

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

The fitness function above is implemented by counting the occurrences of all possible binary substrings from length $h = 1$ up to length N_{max}. In this implementation $N_{max} = 7$, giving us a maximal entropy/fitness of $\sum_{i=1}^{7} i = 28$.

Occurrences are counted using using the KMP[1] pattern matching algorithm.

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

[1] Knuth Morris Pratt - finite automaton based pattern matching

Fitness Function - GP & SNGP - Continued

I replaced the initial brute force pattern matching with the KMP pattern matching algorithm which improved the worst-case running time from $O(nm)$ to $O(n+m)$, and greatly improving running times for both implementations.

After the occurrences are counted for that particular substring length, they are then summed;

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

Fitness Function - GP & SNGP - Continued

The probability of occurrence for each $0 < i \leq H_{max}$ and therefore the scalar entropies can be calculated;

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

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

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

Selection GP

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

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

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

```
for (s = 0; s < POP_SIZE; s++) {
    struct population_member *progp = &population[s];
    acc_fitness = acc_fitness + 1 + progp->total_fitness;
    if (acc_fitness >= selection && selection >= old_acc_fitness) {
        break;
    }
}
```

In the other cases, a parent is chosen at random from the population size (with equal probability).

Selection SNGP

A node is selected at random with equal probabilities, removing the terminal values from the equation. The successor (left or right child) is also chosen with equal probability. This is then mutated to another node in the graph (with a lower depth, chosen with equal probability).

```
int mut = (rand()%(POP_SIZE-NUM_TERMINALS)) + NUM_TERMINALS;
int suc = rand()%2;
```

SNGP Driving Factor

In GP, the evolution of the population is done regardless to fitness progress. In SNGP the most effective climbing approach I found was to drive on either the increase of the total population fitness OR by the fittest members fitness increase;

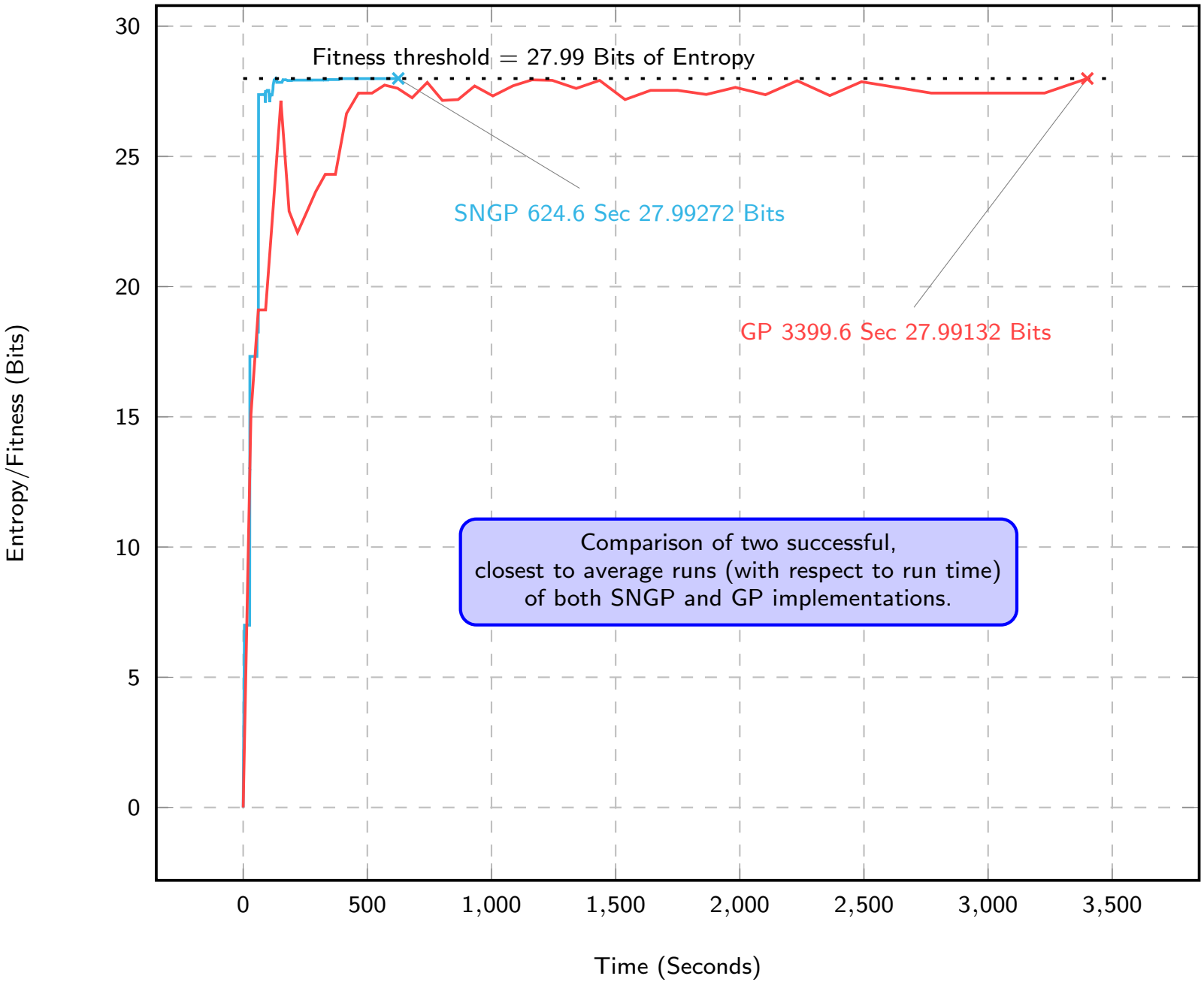
```
if (best_fitness_old >= best_fitness &&
    total_pop_fitness_old >= total_pop_fitness)
{
 //Then throw away new graph population after mutation
}
```

Analysis Snapshot

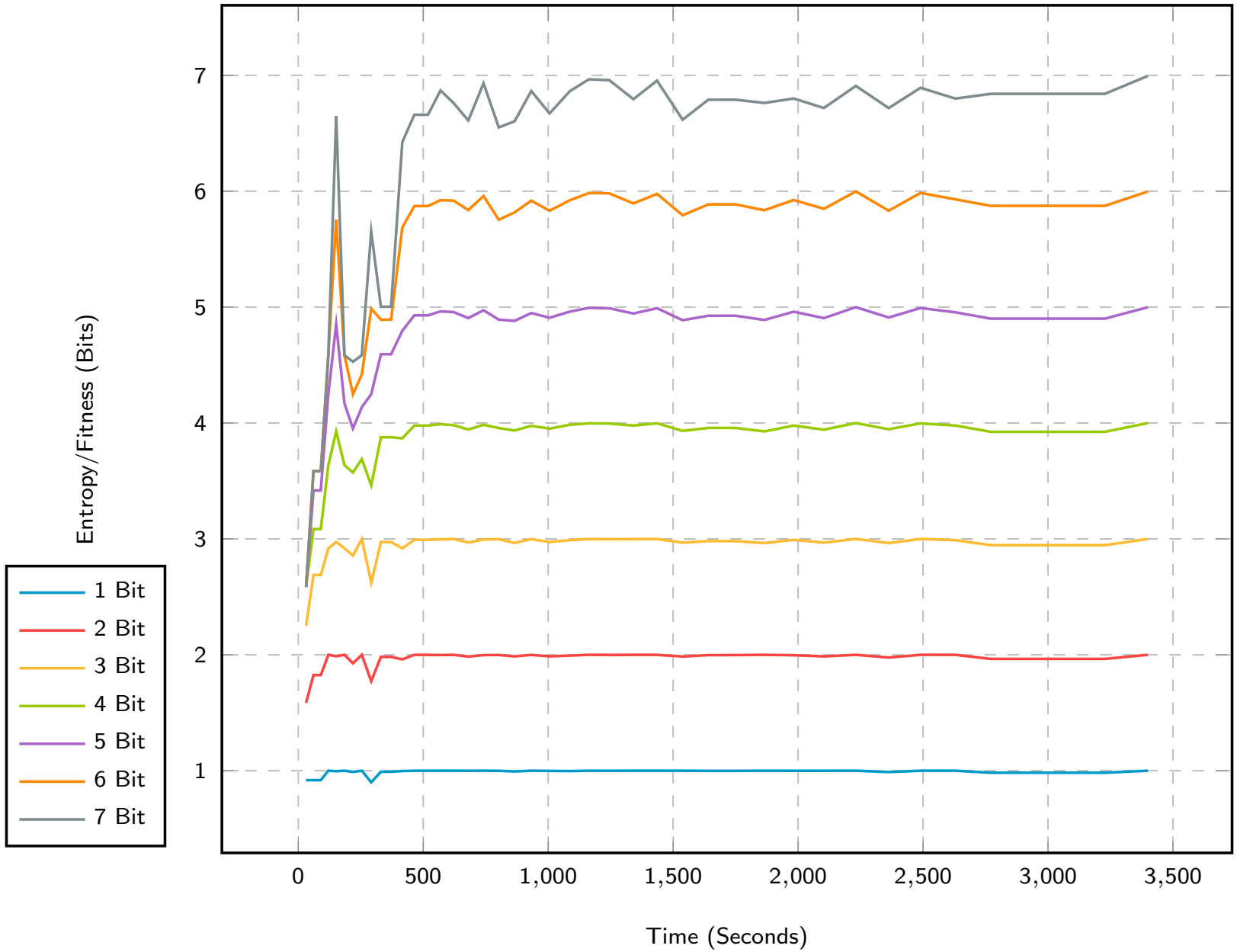
In this section we shall be taking a look at a short but general analysis of the project so far. I conducted 50 runs of both the GP and SNGP implementations in order to gather data to conduct some evaluation of the project. These runs were all conducted under "standard" parameters as seen below.

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

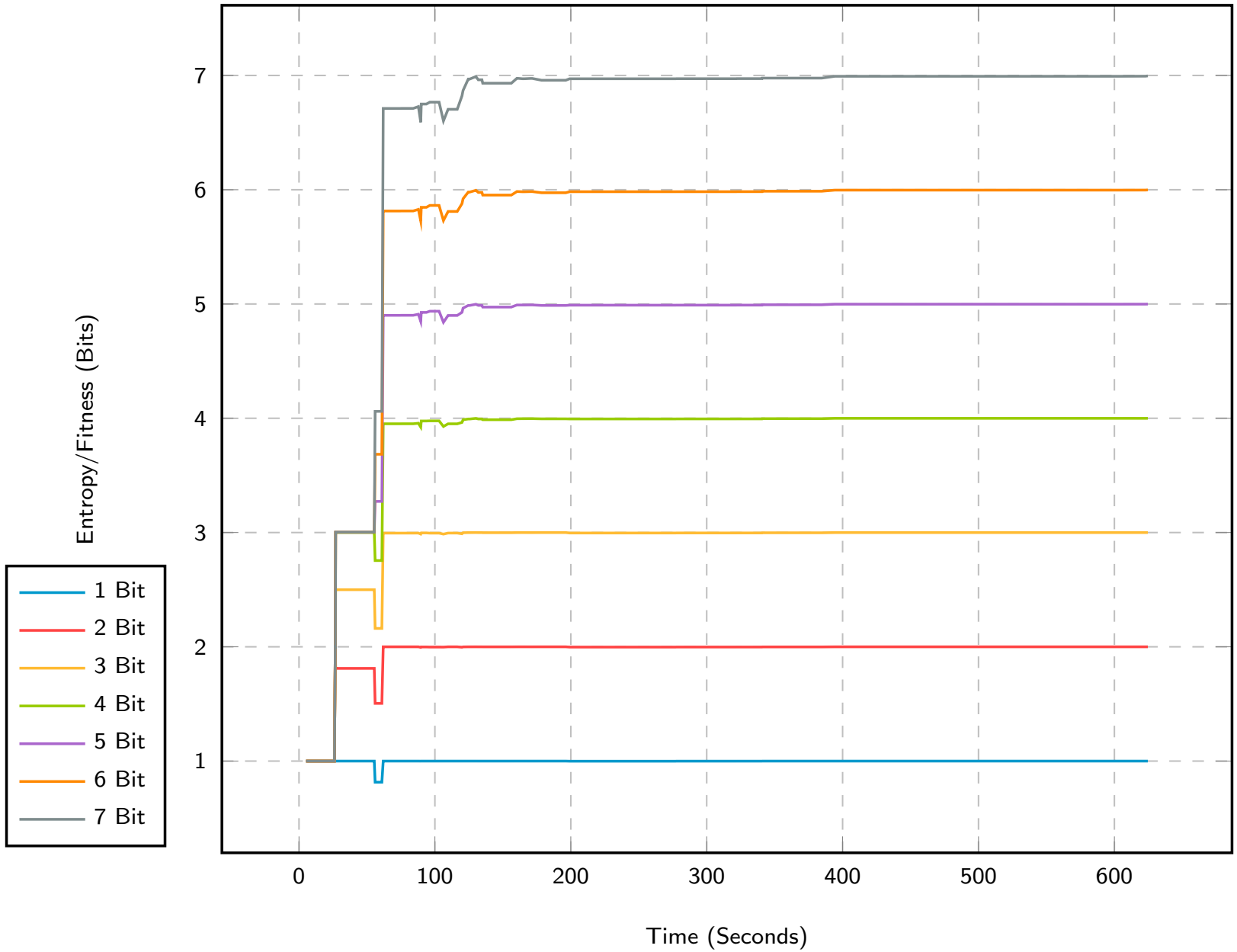
Sample GP vs SNGP - A single Run



Sample GP Scalar Entropies



Sample SNGP Scalar Entropies



50 Run Average GP vs SNGP

After 50 Runs of both implementations, I was able to gather some promising data. The results show that SNGP is both quicker and more effective than GP when it comes to finding solutions (I.e. RNGs with entropy higher than 27.99)

	SNGP	GP
Avg Run Time	606.9s	4111.6s
Avg Sol Run Time	483.2s	2173.4s
Solution Rate	88%	40%
Avg Sol Size	167.5	154.6
Avg Final Size	167.3	277.8
Avg Run Fitness	27.88153318	27.2570626
Best Sol Fitness	27.995729	27.995032
Avg Sol Fitness	27.9917378	27.99195145

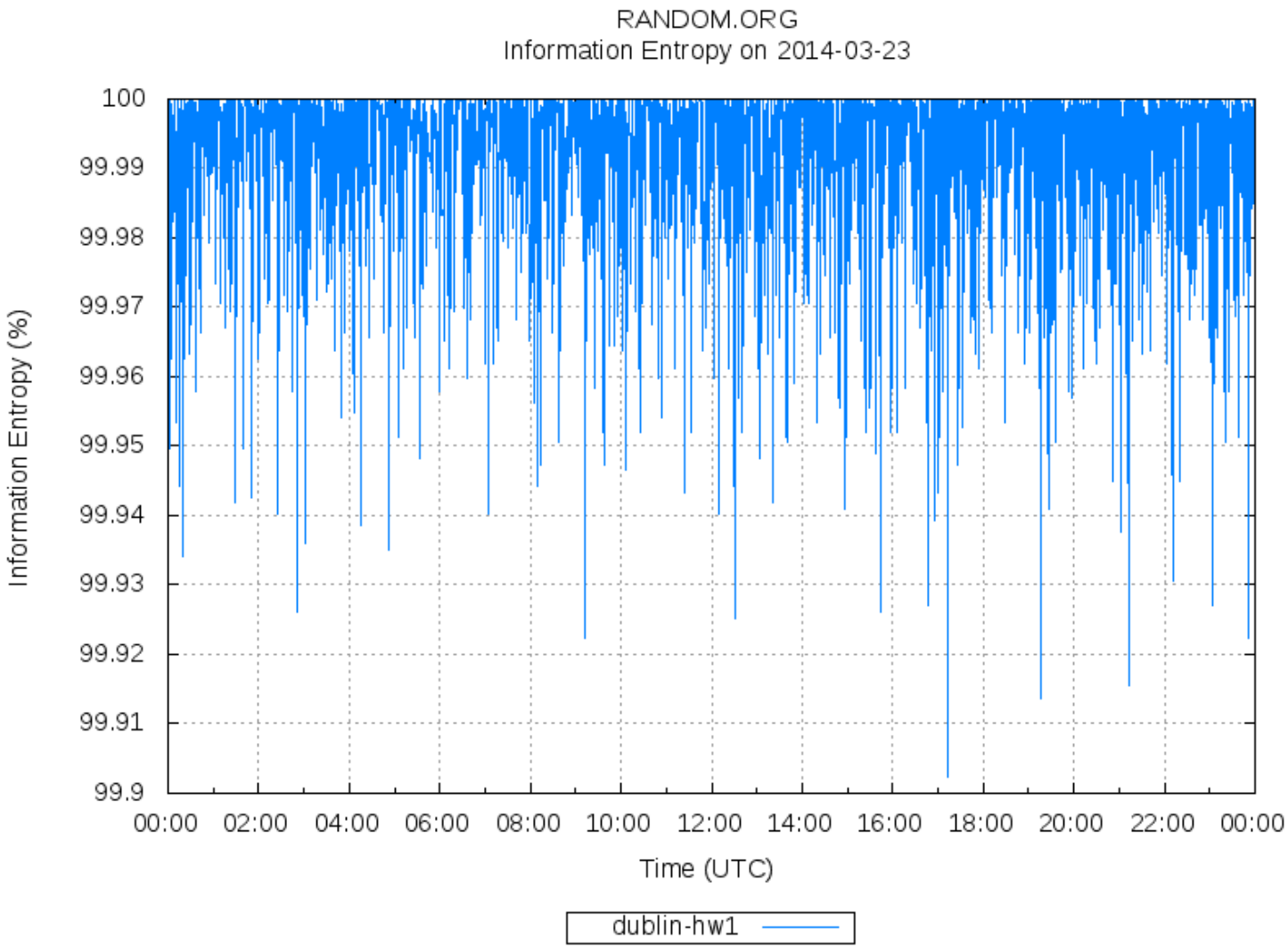
50 Run C Rand() vs Random.org vs GP vs SNGP

I also collected data from 50 calls of both the C Rand function (seeded with the time), and atmospheric noise data from Random.org. The results are promising, showing that the RNGs produced genetically fractionally outperform both common methods of producing random numbers.

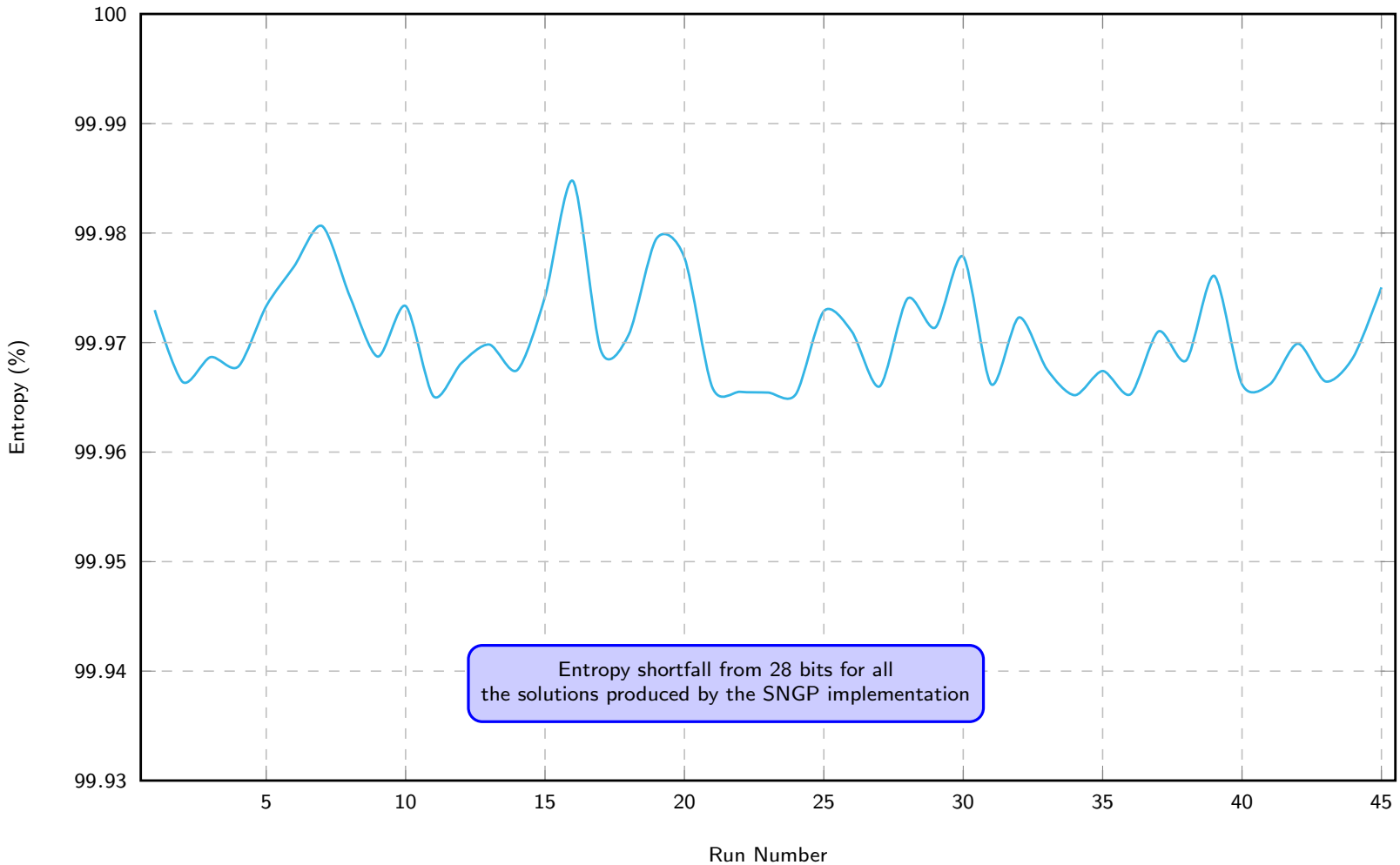
	TRNG	C PRNG	SNGP	GP
Fittest (Entropy)	27.993097	27.991925	27.995729	27.995032
Fittest (%)	99.975%	99.9712%	99.985%	99.982%
Fittest (Entropy Shortfall)	0.006903	0.008075	0.004271	0.004968
Avg Fit (Entropy) (Sol)	27.98695522	27.98985372	27.9917378	27.99195145
Avg Fit (%) (Sol)	99.953%	99.964%	99.970%	99.971%
Avg Fit (Shortfall) (Sol)	0.01304478	0.01014628	0.0082622	0.00804855

Random.org Entropy Fluctuation Over Time

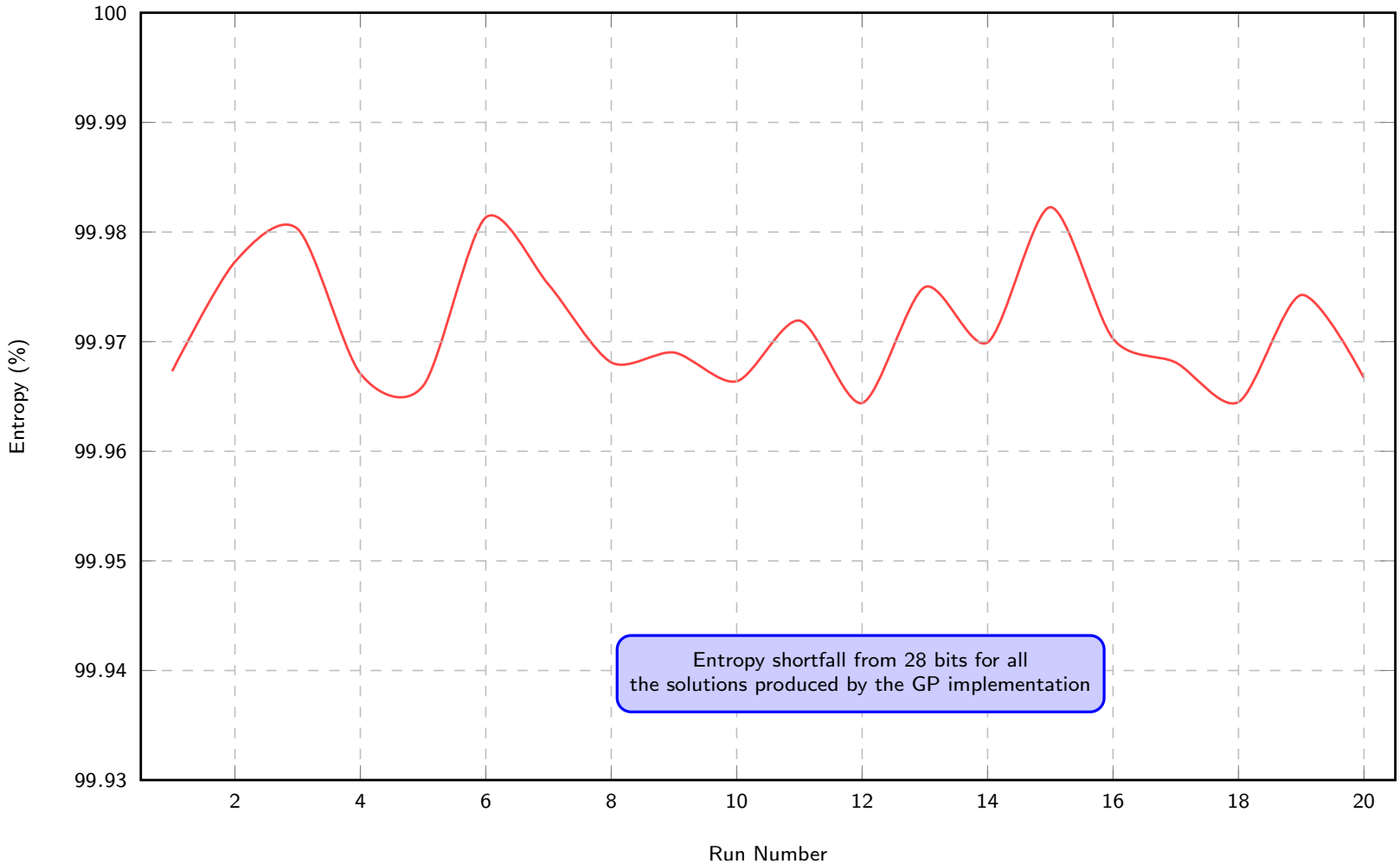
The entropy fluctuation on the day which I gathered the data from Random.org is shown on the graph below generated by their servers.



SNGP Entropy Shortfall of Solutions

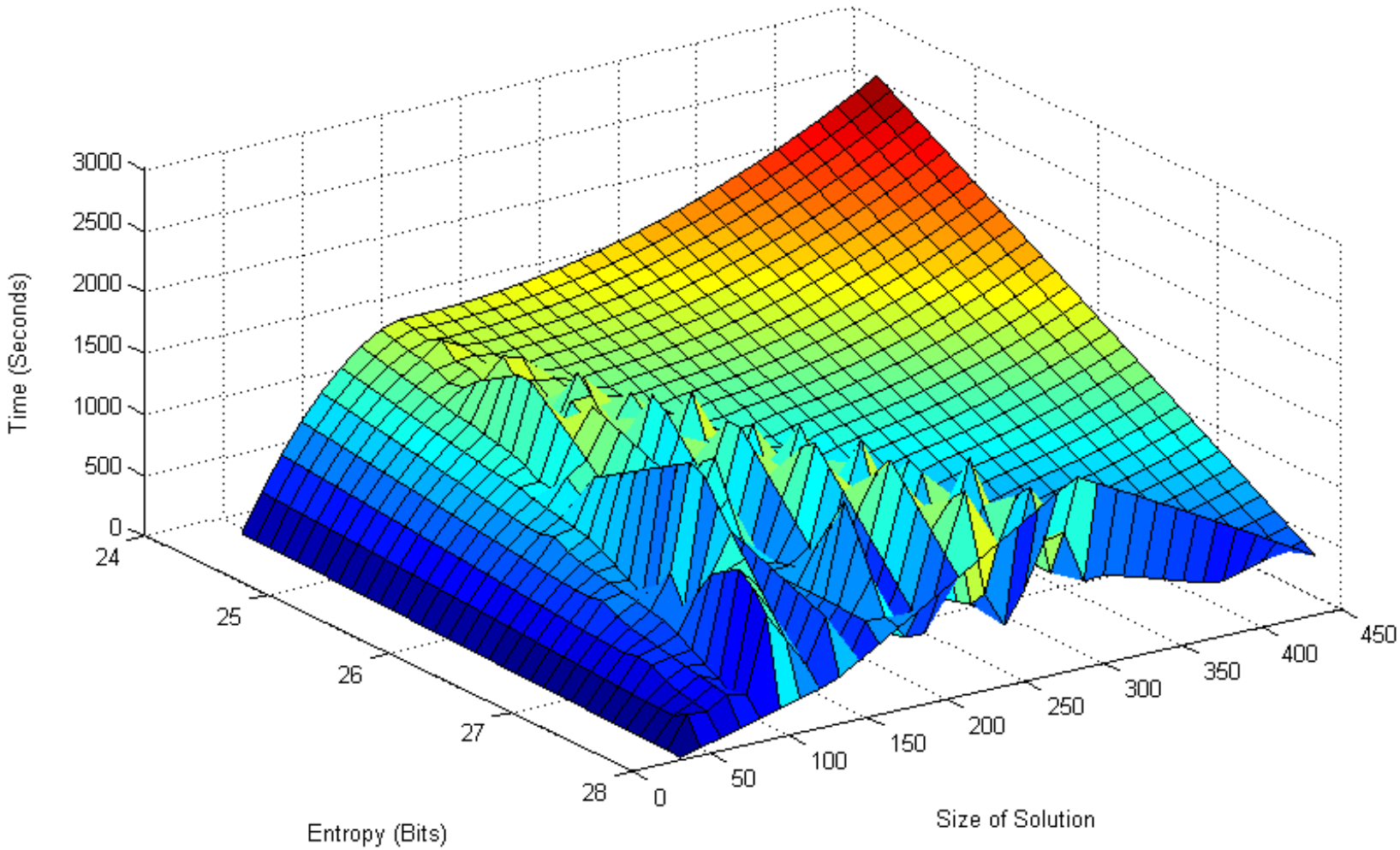


GP Entropy Shortfall of Solutions



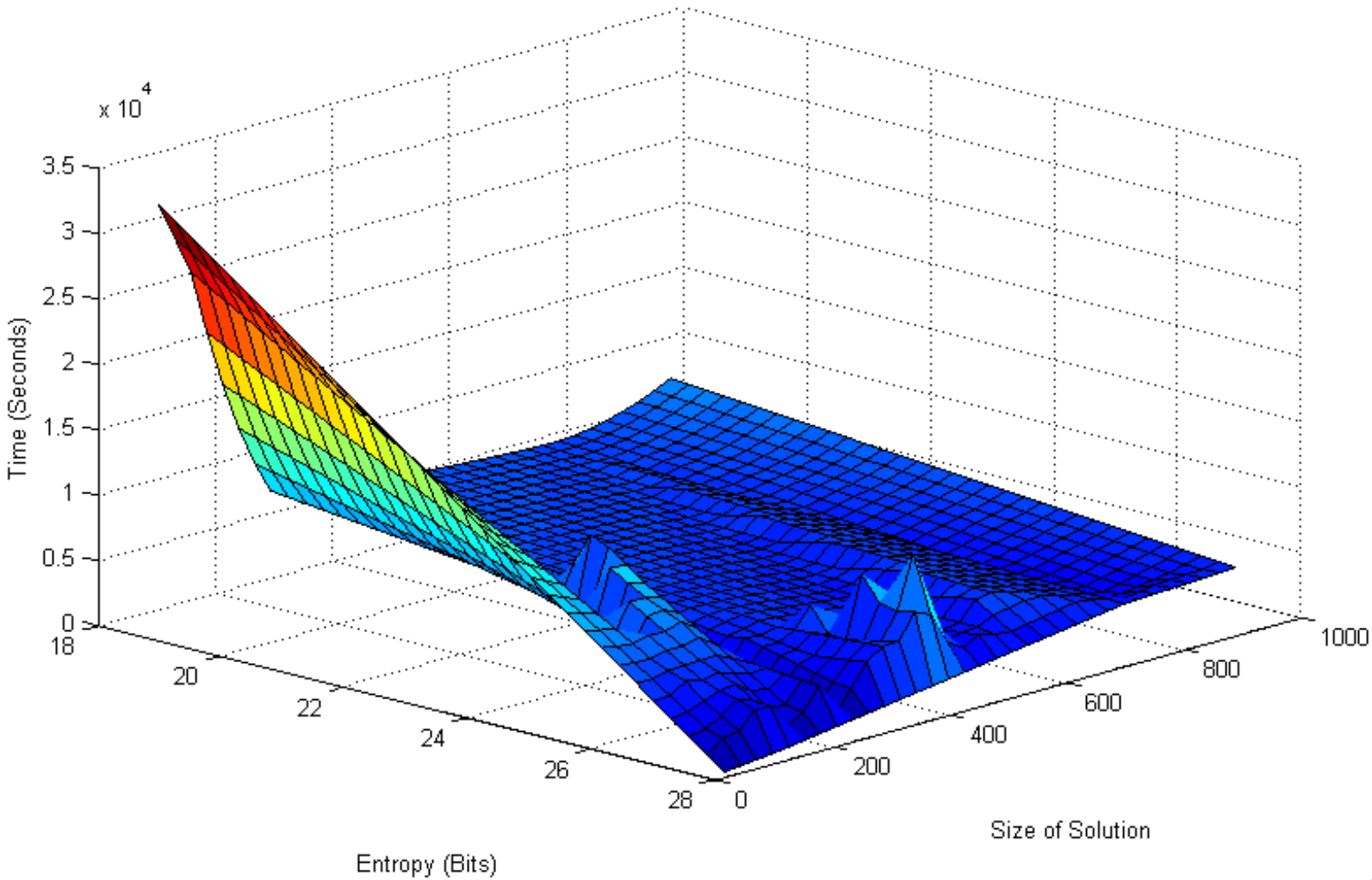
SNGP Surface Plot

A graphical view of the correlation between running time, solution size and Entropy of all 50 runs of the Single Node Genetic Programming Implementation.



Genetic Programming Surface Plot

A graphical view of the correlation between running time, solution size and Entropy of all 50 runs of the Genetic Programming Implementation.



Overview

To conclude, it is clearly evident that SNGP produces more solutions in a much quicker time frame, under these standard parameters. It is also evident that a RNG produced genetically can outperform common pseudo and natural methods of producing random numbers.

This has just been a snapshot of the evaluation, in my dissertation further analysing of both GP and SNGP will be undertaken using different parameters in order to observe performance under different conditions.

Progress

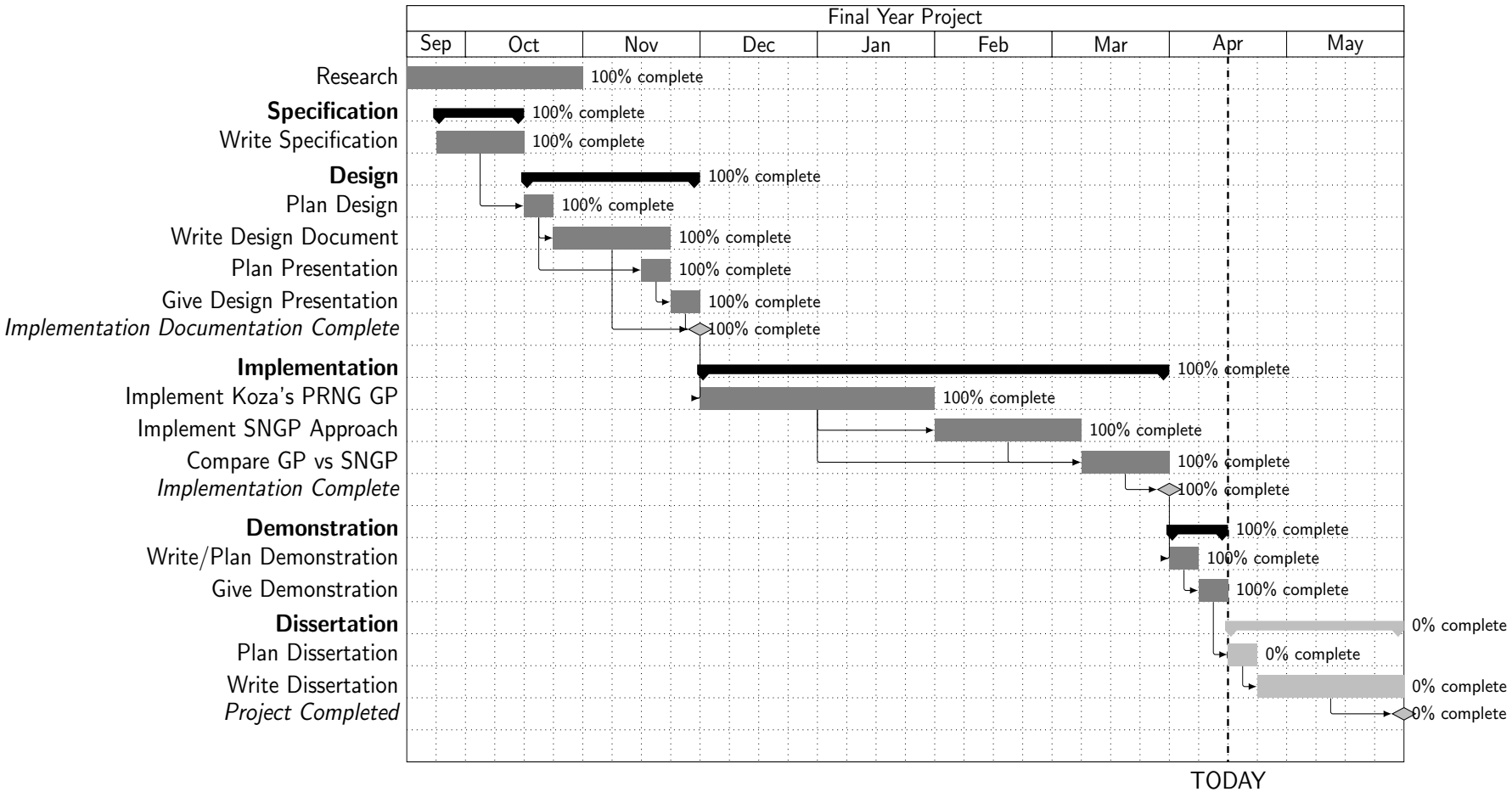


Figure: Gantt Chart - Work plan


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


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

Q&A