On the Design of State-of-the-Art Pseudorandom Number Generators by Means of Genetic Programming

Julio Cesar Hernández, Andre Seznec
CAPS Team
INRIA-IRISA
Campus de Beaulieu
35042 Rennes, France
Email:{jcesar, seznec}@irisa.fr

Pedro Isasi
Computer Science Department
Carlos III University
28911 Leganés, Madrid, Spain
Email: isasi@ia.uc3m.es

***Abstract*- The design of pseudorandom number generators by means of evolutionary computation is a classical problem. To day, it has been mostly and better accomplished by means of cellular automata and not many proposals, inside or outside this paradigm could claim to be both robust (passing all the statistical tests, including the most demanding ones) and fast, as is the case of the proposal we present here. Furthermore, for obtaining these generators, we use a radically new approach, where *our fitness function is not at all based in any measure* of randomness, as is frequently the case in the literature, but of non-linearity. Efficiency is assured by using only very efficient operators (both in hardware and software) and by limiting the number of terminals in the Genetic Programming implementation.**

I. Introduction

The relation between pseudorandom number generators (PRNGs) and evolutionary computation techniques has been long and fruitful. It could be divided in two major areas: the first and more popular has as scope trying to use evolutionary computation techniques to develop good PRNGs. In this area, literally hundreds of works have been published and virtually all EC paradigms have been used, ranging from genetic algorithms [6] to genetic programming [12], including cellular automata [31] (and cellular programming [28]) and ant colony systems [7, 8, 9], to name only a few.

Results obtained diverge very much in its quality and applicability, as they typically cannot pass a battery of very demanding statistical tests or, if they can, it is usually at a high cost to its efficiency and, hence, to its applicability. This seriously limits the practical implications of this field of research, with the notable exception of Cellular Automata, where some recent papers [25, 26] have shown that some models are able to produce an output that pass the most difficult tests batteries while being very efficient and quite adequate to be implemented in hardware.

The second approach, more recent and that could probably lead in the near future to very interesting results, is to use the behavior of EC algorithms to measure the quality of PRNGs. Some quite promising papers [1, 21, 22, 23] have been recently published in this area.

However, in this work we will focus on the first subject area, and will present a novel proposal that, using a paradigm different to Cellular Automata, is able of producing both very efficient and high quality PRNGs, which are able of passing even the more hard tests, including some that have recently been published and are considered very difficult. What is even more important, we will do so by using a completely new strategy.

A. Fitness Functions and Randomness

The most common approach in the literature seems to be to use as a fitness function, for whatever the EC algorithm, some measure of the randomness properties of the output. In this vein, the result of one (i.e. entropy per byte as in [6]) or, more usually, many, (i.e. the FIPS-140 battery, as in [7]) statistical tests are used to assign a given fitness to an individual, typically mixing these results in a mathematical expression (possibly assigning certain weights to the different tests, depending on its significance, etc.).

Other possibility also found in the literature [26] is to use MOO (Multi Objective Optimization) over the results of these tests.

The common problem to these approaches is that the generators obtained need not to pass any other tests than those that form part (or other, quite related, statistical independence does not necessary hold between randomness tests) of the fitness function, and this is frequently the case: the generators produce, naturally, nearly optimal values for all the tests included in the fitness function but quickly fail other, not related, previously unseen, tests. Even very simple ones. It is true

0-7803-8515-2/04/$20.00 ©2004 IEEE

that, sometimes, the aforementioned strategy seems to work, but in most of the cases it does not.

That's why we propose a completely new strategy, which does not include any kind of randomness measure in the fitness function (such as entropy, serial correlation coefficient, average, etc.) but a measure of non-linearity. This change is quite important, because randomness has not a clear definition, depends of the observer, the tests used, the applications where one will use the generators, and there are multiple definitions for the concept which not satisfy all authors and which, more importantly, make it very difficult, if not impossible, to obtain an undisputed measure (excellent discussions on this topic are presented in [2, 10, 11, 12]).

Even the quite intuitive term of entropy is not enough to cover the randomness concept, as it is possible to be presented with generators with exceptionally high output entropy [6, 12] which are far from random.

That is the reason why we prefer to measure something that has a clear definition, that is univocally measurable, that could not mislead us, and our proposal is to measure one mathematical property of mathematical functions that is, in some way, a measure of non-linearity. It is called the avalanche effect.

B. The Avalanche Effect and Strict Avalanche Criterion

Similarly to randomness, nonlinearity can be measured in a number of ways or, what is equivalent, has not a complete unique and satisfactory definition.

Fortunately, this is of no concern to us as we do not pretend to measure non-linearity but a very specific mathematical property named avalanche effect. This property tries to reflect, to some extend, the intuitive idea of high-nonlinearity: a very small difference in the input producing a high change in the output, thus an avalanche of changes. Mathematically, $F : 2^m \rightarrow 2^n$ has the avalanche effect if and only if it holds that:

$$x, y \mid H(x, y) = 1 \; Avrg(H(F(x), F(y))) = \frac{n}{2}$$

So if F is to have the avalanche effect, the Hamming distance between the outputs of a random input vector and one generated by randomly flipping one of the bits should be, on average, *n/2*.

That is, a minimum input change (one single bit) produces on average a maximum output change (half of the bits). This definition also tries to abstract the more general concept of independence of the output from the input (this justifies our proposal and its applicability to the generation of good PRNGs). Although it is clear that this independence is impossible to achieve (a given input vector always produces the same output) the ideal F will resemble a perfect random function where inputs and outputs are statistically unrelated. Any such F would have perfect avalanche effect, so it is natural to try to obtain such functions by optimizing the amount of avalanche. In fact, we will use an even more demanding property that has been called the Strict Avalanche Criterion [3] which, in particular, implies the Avalanche Effect, and that could be mathematically described as:

$$\forall x, y \mid H(x, y) = 1 \; H(F(x), F(y)) \approx B(\frac{1}{2}, n)$$

It is interesting to note that this implies the avalanche effect, because the average of a Binomial distribution with parameters *1/2* and *n* is *n/2*, and that the amount of proximity of a given distribution to a certain distribution (in this case a Binomial *B(1/2,n)*) could be easily measured by means of a chi-square goodness-of-fit test. That is exactly the procedure we will follow.

II. Genetic Programming

Genetic Programming [15] is a method for automatically creating working computer programs from a set of high-level statements of a given problem.

This is achieved by breeding a population of computer programs using the principles of Darwinian natural selection and other biologically inspired operations that include reproduction, sexual recombination (crossover), mutation, and possibly others.

Starting from an initial population of randomly created programs derived from a given set of functions and terminals, populations gradually evolve, giving birth to new, more fitted individuals.

This is performed by repeating the cycle of fitness evaluation, Darwinian selection and genetic operations until a certain ending condition is met. Each individual (or program in the population) is evaluated to determine how fit is at solving a given problem, and then programs are selected probabilistically from the population according to their fitness values for being applied the rest of genetic operators.

It is important to note that, while fitter programs have higher probabilities of being selected, all programs have a chance. After some generations, a program may emerge that solves, completely or approximately, the problem at hand.

Genetic Programming combines the expressive high-level symbolic representations of computer programs with the learning efficiency of genetic algorithms. Genetic Programming techniques have been successfully applied to a number of different problems: apart from classical problems such as function fitting or pattern recognition, where other evolutionary computation techniques also work fine, they have even produced results that are competitive with humans in some non-trivial tasks as designing electrical circuits [14] (some of which have been patented) or at classifying protein segments [13].

III. Implementation Issues

We have used the lilgp genetic programming system [34], but a number of modifications were needed for our problem.

Firstly, we need to define the set of functions: This is critical for our problem, as they are the building blocks of the algorithms we would obtain. Being efficiency one of the paramount objectives of our approach, it is natural to restrict the set of functions to include only very efficient operations, both easy to implement in hardware and software. Another, but minor, objective was to produce portable algorithms; so the inclusion of the basic binary operations as **rotd** (right rotation), **roti** (left rotation), **xor** (addition mod 2), **or** (bit wise or), **not** (bit wise not), and **and** (bit wise and) are an obvious first step. Other operators as the **sum** (sum mod 2^{32}) are necessary in order to avoid linearity, being itself quite efficient.

Another interesting operator introduced was **kte**, an operation that, whatever its inputs, returns the 32-bit constant value 0x9e377969 (which are the most significant digits of the expression of the golden ratio in hexadecimal notation). The idea behind this operator was to provide a constant value that, independently from the input, could be used by the aforementioned operators to increase non-linearity, and idea suggested by [30].

The inclusion of the **mult** (multiplication mod 2^{32}) operator was not so easy to decide, because, depending on the particular implementations, the multiplication of two 32 bit values could cost up to fifty times more than an **xor**, a **rotd** or an **and** operation (although this could happened in certain architectures, it's nearly a worst case: 14 times [5] seems to be a more common value), so it is relatively inefficient, at least when compared with the rest of the operators used. In fact, we did not include it at first, but after extensively experimentation, we conclude that its inclusion was beneficial because, apart from improving non-linearity; it at least doubled and sometimes tripled the amount of avalanche we were trying to maximize, so we finally introduce it in the function set.

Equally, after many experiments, we concluded that the functions **roti** and **rotd** where absolutely interchangeable and not necessary, nor useful, at the same time, so we arbitrarily decided to remove **roti** and left **rotd**. We also used ephemeral constants. The set of terminals in our case is easy to establish, as the input will be exclusively formed by one 32 bits integer $\mathbf{a_0}$, and it will be at the branches of the function trees that the genetic programming algorithm will construct, with functions from the function set in the nodes.

The fitness of every individual (algorithm or function) was evaluated by generating 1024 32-bit random vectors (using the Mersenne Twister generator [19]), then randomly flipping one of the bits and calculating the Hamming distance over their outputs.

For each of these 1024 experiments a Hamming distance between 0 and 32 was obtained and stored. The fitness of the individual (or function) under observation was proportional to the inverse of the value of the chi-square statistic that measured the distance from the optimal probability distribution (the *B(1/2,32)*) of the observed distribution of these Hamming distances. Thus

$$Fitness(I) = \frac{1000}{\sum_{h=0}^{h=32} \frac{(O_h - E_h)^2}{E_h^{\,2}}}$$

where $E_k = 1024 * \Pr(B(1/2,32) = k)$ and we are calculating the value of the chi-square statistic without the commonly used restriction of adding up only the values when $E_k > 5.0$ for amplifying the effect of a bad output distribution, thus, the sensibility of our measure.

When using genetic programming approaches, it is necessary to put some limits to the depth and to the number of nodes the resulting trees could have. We tried various ideas here, both limiting the depth and not limiting the number of nodes and vice versa and the best results where consistently obtained using this latter option, so we fixed the number of maximum nodes to 20 and did not put a limit (other that the number of nodes itself) to the tree depth. This is also an important step for assuring the efficiency of the resulting algorithm.

We selected a population size of 100 individuals, a crossover probability of 0.8, which produced better results

than the default 0.9 probability proposed, and an ending condition of reaching 10000 generations. Ten different runs were performed, each one seeded with the 6 most significant digits of the expression $(314159)^{i=0..9}$ and the best results achieved after every run are shown in Appendix A.

IV. Results

From the different 10 individuals obtained after each run, we selected the best three (in terms of their fitness functions, in bold in Appendix A) and calculated their simplified expressions. We then combined (á la KISS [17]) these three PRNGs to form a new and better PRNG, that we will afterwards call genbgen, and used a new and very demanding battery of tests to evaluate the performance of the new PRNG.

This battery of tests is claimed to be harder to pass than the classical DIEHARD [16] test suite, and includes some improved and harder versions of tests already present in DIEHARD, toghether with some new tests. We have used the implementation in [33]. The authors say [18] that every generator they had tested that passes the three hard tests in the battery (GCD test, Birthdays spacing test and Gorilla test) not only passes all tests in the DIEHARD battery but all statistical tests they know of.

It is important to note that this was the first time our generator (our any of their components) where tested for randomness. The results over ten different runs are shown in Table I. The three seeds needed for each run were obtained in Random.org [32]

TABLE I : THE RESULTS IN EACH TEST SHOULD FOLLOW A UNIFORM DISTRIBUTION IN THE (0,1) INTERVAL.

Exp. #	p-value GCD	p-value Gorilla	p-value Birthday
1	0.643302	0.0849	0.1432
2	0.721130	0.4029	0.2524
3	0.831144	0.3723	0.5033
4	0.781355	0.0625	0.2628
5	0.548920	0.1888	0.9962
6	0.346309	0.8267	0.2828
7	0.086146	0.5674	0.0724
8	0.602264	0.8114	0.5268
9	0.853532	0.3530	0.7278
10	0.401882	0.1882	0.8712

These results clearly show that the proposed generator passes all the tests in the battery (generators that do not pass the tests produce values too close to 0 or 1, up to 6 or more digits.), which is a very important result as many widely used, well known generators fail to pass these tests [18]. Additionally, we have tested our generator against some other well-known strict tests commonly used in the literature [31]. Results are shown in Table II.

TABLE II : THE RESULTS IN EACH TEST SHOULD FOLLOW A UNIFORM DISTRIBUTION IN THE (0,1) INTERVAL.

Exp. #	p-value Frequency test	p-value Collisions test
1	0.537453	0.5813
2	0.242357	0.9785
3	0.074201	0.7452
4	0.687466	0.8290
5	0.869928	0.6473
6	0.043511	0.8890
7	0.942138	0.9099
8	0.621991	0.1414
9	0.307241	0.6765
10	0.708462	0.4848

V. Conclusions

From the results shown in Table I and II, we can conclude that the proposed generator is able of passing one of the most demanding battery of statistical tests and some other quite strict tests, so its output should be considered as having excellent pseudorandom properties, better than many other commonly used PRNGs. Our proposal, at the same time, has the advantage of being portable, efficient, highly parallelizable and compact.

We have also shown that a completely new strategy for generating pseudorandom number generators by means of evolutionary computation could lead to very interesting results, a strategy that does not need in any way to evaluate the randomness (if this could be done, anyway) of the individuals but only evaluate the proximity of a given function to a certain, well-defined, mathematical property.

Although not being able of distinguishing the output of our proposed generator (source code in C in Appendix B), it is clear that we cannot recommend it for cryptographic use. It is obvious that it presents some undesirable properties for cryptography, and, for example, it should be seeded with random numbers because otherwise (i.e. null seeds) the output of the generator could present very

bad statistical properties. This, however, is quite a common feature of many PRNGs, as, for example, in LFSRs [4].

Finally, we should conclude that the output of the proposed generator, as proved by passing and performing remarkably well in all the statistical tests examined, offers enough quality to be used in a variety of scientific, mathematical, engineering and industrial applications, including Montecarlo simulations, decision theory, game theory, sampling, etc.

Random numbers, and, extensively, random number generators, are very difficult to create [Park98, Vattu95] but in this paper we have shown a completely new approach that could significantly aid in easing, to the point of making it automatic, this process.

ACKNOWLEDGMENTS

Supported by the Spanish Ministerio de Ciencia y Tecnologia Research Project TIC2002-04498-C05-4 and by the INRIA Postdoctoral Fellowship program.

REFERENCES

[1] Cantú-Paz, E. "On Random Numbers and the Performance of Genetic Algorithms." *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, p.311-318

[2] Chaitin, G.J. *Information, Randomness & Incompleteness.* World Press Publishing, Singapore, 1987

[3] Forre R. "The strict avalanche criterion: spectral properties of Boolean functions and an extended definition." *Advances in Cryptology, CRYPTO 88, Lecture Notes in Computer Science*, vol. 403, S. Goldwasser ed., Springer-Verlag, pages 450-468, 1990

[4] Golomb, S. *Shift Register Sequences.* Aegean Park Press, Laguna Hills, CA, 1982.

[5] Hinton, Glenn et al. "The micro architecture of the Pentium 4 processor. " *Intel Technology Journal* Q1, 2001.

[6] Hernandez, J., C., Ribagorda, A., Isasi, P. and Sierra, J., M. "Finding near optimal parameters for Linear Congruential Pseudorandom Number Generators by means of Evolutionary Computation." *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* p. 1292-1298. Morgan-Kauffmann, 2001

[7] Isaacs, J., C., Watkins, R., K. and Foo, S., Y. "Evolvable Ant Colony Systems for Pseudo-Random Number Generation." *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* p. 770, Morgan-Kauffmann, 2001

[8] Isaacs J.C., Watkins R.K.and Foo S.Y. "Evolvable Random Number Generators: A Few Ants in Your Hardware could be a Good Thing." *Proceedings of the 2002 MAPLD International Conference.*

[9] Isaacs, J., Watkins, R. and Foo, S. "Evolving Ant Colony Systems in Hardware for Random Number Generation." *Proceedings of the Congress on Evolutionary Computation CEC2002*, p. 1450-1455

[10] Knuth, D. E., 1997, *The Art of Computer Programming*, Vol. 2, 3rd ed., Addison-Wesley

[11] Kolmogorov, A. "Three approaches to the quantitative definition of information" in *Problems of Information Transmission* v.1 n.1. Faraday Press, NY, 1965

[12] Koza, J. "Evolving a computer program to generate random number using the genetic programming paradigm." *Proc. of the Fourth Int. Conf. on Genetic Algorithms*, Morgan Kauffman, pp. 37-44, 1991

[13] Koza, J., Andre, D. "Automatic discovery of protein motifs using genetic programming." In *Evolutionary Computation: Theory and Applications.* World Scientific Publications, 1996

[14] Koza, J., et al. Automated "Synthesis of analog electrical circuits by means of genetic programming." In *IEEE Transactions on Evolutionary Computation* 1(2), 1997

[15] Koza, J.: "Genetic Programming." In *Encyclopedia of Computer Science and Technology*, v.39, 29-43, 1998

[16] Marsaglia, G., 1995, Diehard battery of tests of randomness, The Marsaglia random number CDROM, Department of Statistics, Florida State University.

[17] Marsaglia, G. "Random number generators for C: Some suggestions. " Posting in newsgroups *sci.math*, January 1999.

[18] Marsaglia, G. and Tsang W.W. "Some difficult to pass tests." *Journal of Statistical software*, Volume 7, 2002, issue 3.

[19] Matsumoto M and Nishimura T., "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator" *ACM Trans. on Modeling and Computer Simulation* Vol. 8, No. 1, January pp.3-30 1998

[20] Maurer, U., 1992, "A universal statistical test for random bit generators", *Journal of Cryptology* 5, No. 2, 89-105.

[21] Meysenburg, M., M. and Foster, J., A. "The Quality of Pseudo-Random Number Generators and Simple Genetic Algorithm Performance." *Proc. of the Seventh Int. Conf. on Genetic Algorithms* p. 276-282, 1997

[22] Meyesenburg, M., M., Hoelting, D., McElvain, D. and Foster, J., A . "A Genetic Algorithm-specific Test Of Random Generator Quality." *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, p. 691

[23] Meyesenburg, M., M., Hoelting, D., McElvain, D. and Foster, J., A "How Random Generator Quality Impacts GA Performance." *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* p. 480-487

[24] Park S.K. and Miller K.W. "Random number generators: Good ones are hard to find." *Communications of the ACM*, v. 31, pags. 1192-1201, 1988

[25] Shackleford B., Tanaka M., Carter R.J., Snider G. "FPGA implementation of neighborhood-of-four cellular automata random number generators" in *Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field-Programmable Gate Arrays*, pgs. 106 – 112, 2002

[26] Sheng-Uei and Shu Zhang. "An evolutionary approach to the design of controllable cellular automata structure for random number generation." *IEEE Transactions on Evolutionary Computation*, Vol. 7, N. 1, February 2003.

[27] Sipper M. and Tomassini M.. "Co-evolving parallel random number generators." *Proceedings of the Parallel problem solving from Nature-PPSN IV*, pages 950-959, 1996, Springer-Verlag

[28] Sipper, M. and Tomassini, M. "Generating parallel random number generators by cellular programming" *International Journal of Modern Physics C*, p. 181-190, 1996

[29] Vattulainen I., Ala-Nissila T. "Mission impossible: Find a random pseudorandom number generator." *Computers in Physics*, September 1995.

[30] Wheeler, D., Needham, R.: "TEA, a Tiny Encryption Algorithm." In *Proceedings of the 1994 Fast Software Encryption Workshop*

[31] Wolfram S. "Random Sequence Generation by Cellular Automata." *Advances in Applied Mathematics*, 7. June 1986. pgs. 123-169

[32] Random.org. The true random number service http://random.org/cgi-bin/randbyte?nbytes=256&format=hex

[33] Center for Information Security and Cryptography (CISC) Library of Tests for Random Number Generators at http://www.csis.hku.hk/cisc/download/idetect/

[34] The lilgp genetic programming system is available at http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html.

APPENDIX A

These are the three best individuals found after ten rounds, used in the genbgen pseudorandom number generator:

```
=== BEST-OF-RUN ===
        generation: 48 nodes: 20 depth: 15
TOP INDIVIDUAL:
hits: 139091 raw fitness: 1137.5078
standardized fitness: 0.8791 adjusted fitness: 0.8791
TREE:
 (sum (mult a0 ecb8462d)
      (rotd (rotd (rotd (rotd (mult (rotd (rotd (rotd (rotd
(rotd (rotd (rotd (
 rotd (rotd a0)))))))) ecb8462d))))))

=== BEST-OF-RUN ===
        generation: 76 nodes: 20 depth: 16
TOP INDIVIDUAL:
hits: 130468 raw fitness: 2456.0547
standardized fitness: 0.4072 adjusted fitness: 0.4072
TREE:
 (mult (xor a0
          (rotd (rotd (rotd (rotd (rotd (rotd (rotd (rotd
(rotd (rotd (rotd (
 rotd (rotd (rotd a0))))))))))))))
     (kte a0))

=== BEST-OF-RUN ===
        generation: 321 nodes: 20 depth: 17
TOP INDIVIDUAL:
hits: 145664 raw fitness: 1260.9751
standardized fitness: 0.7930 adjusted fitness: 0.7930
TREE:
 (mult 81d6cf85
      (rotd (rotd (rotd (rotd (rotd (rotd (rotd (rotd (rotd
(rotd (rotd (rotd
 (rotd (rotd (rotd (mult a0 81d6cf85)))))))))))))))))
```

APPENDIX B

Genbgen generator source code in C

```
/* The genbgen generator. The name stands for
GENetically Born GENerator. Developed by Julio C.
Hernandez, Isasi P., and Seznec, A.        */

unsigned long genbgen(unsigned long seed1,seed2,seed3)
  {
  //initializations
  static unsigned long r1=seed1,r2=seed2,r3=seed3, r2p;

  r1=r1*0x81d6cf85; r1=r1>>15; r1=r1*0x81d6cf85;

  r2=r2p;r2=(r2>>9)*0xecb8462d;
  r2=(r2>>4)+(r2p*0xecb8462d);

  r3=((r3>>14)^r3)*0x9e3779b9;

  return((r1^r2)+(r3)+(r1^r3)+(r1)+(r2^r3)+(r2));}
```

This code can be executed, for example, with the call genbgen(0xd2df48f5,0x0ae72b13,0xaf7e2238)