Evolving a Computer Program to Generate Random Numbers Using the Genetic Programming Paradigm

John R. Koza Computer Science Department Stanford University Stanford, CA 94305 USA Koza@Sunburn.Stanford.Edu 415-941-0336

ABSTRACT

This paper demonstrates that it is possible to genetically breed a computer program that is considered difficult to write, namely, a randomizer that converts a sequence of consecutive integers into pseudo-random bits with near maximal entropy.

1. INTRODUCTION AND OVERVIEW

"How can computers learn to solve problems without being explicitly programmed?" This question, which is a central question in the fields of artificial intelligence and machine learning, can be approached using an analogy to the evolutionary process in nature.

John Holland's pioneering 1975 Adaptation in Natural and Artificial Systems [3] described how the evolutionary process in nature can be applied to artificial systems using the "genetic algorithm" operating on fixed length character strings.

Representation is a key issue in genetic algorithm work because genetic algorithms directly manipulate the coded representation of the problem and because the representation scheme can severely limit the window by which the system observes its world [2]. For many problems in artificial intelligence, the most natural representation for solutions to problems are hierarchical computer programs of indeterminate size and shape, as opposed to structures whose size has been determined in advance. It is unnatural and difficult to represent hierarchies of dynamically varying size and shape with fixed length character strings.

In this paper, we show how to genetically breed a population of computer programs to convert a sequence of consecutive integers into a sequence of pseudorandom bits using the recently developed "genetic programming" paradigm.

2. DEFINITION OF RANDOMNESS

Numbers "chosen at random" are useful in a variety of scientific, mathematical, engineering, and industrial applications, including Monte Carlo simulations, decision theory, instant lottery ticket production, etc. Random numbers are difficult to create. Marsaglia [18] describes the difficulty of successfully randomizing numbers, particularly where a stream of several "independent" random integers are required to carry out related steps of one process.

When random numbers are required in computer programs, they are typically provided by a deterministic algorithm (as opposed to some non-algorithmic technique, such as neutron emissions). The algorithm is known to the programmer. Moreover, the program is typically written (using "seeds") so that its output is fully reproducible (to aid debugging, verification, etc.). Numbers which are produced by a deterministic, known, and reproducible algorithm are, of course, anything but random. Indeed, as John Von Neumann said, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

Not only is the term "random number" an oxymoron, but there is no generally accepted mathematical definition of a random number sequence. Knuth [17] describes a number of different specific tests, such as the equidistribution (frequency) test, gap test, run test, serial test, permutation test, coupon collector's test, poker test, etc. D.H.Lehmer (17) said "A random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and *depending somewhat on the uses to which the sequence is to be put* ." (Italics added).

The notion of statistical independence can be used as the starting point for one possible rigorous mathematical definition of a sequence $\{u_i\}$ of random numbers. Anderson [1], for example, says that "each value of u_i is as likely as any other value and the value of u_i must be statistically independent of the value of u_j for $i \neq j$." The first part of this statement corresponds to the equidistribution (frequency) test described in Knuth [17] while the second part of this statement corresponds to a pairwise version of the serial test described in Knuth.

One can expand Anderson's conditions to a more general concept of statistical independence based on the intuitive notion of conditional probability that a specified sub-sequence predicts another sub-sequence. Let K be the number of values possible at a given position in the sequence. Let c be the number of consequent (i.e. predicted) positions within a subsequence of length N. Let g (where $g \le N-c$) be the number of antecedent (i.e. given) values at the remaining N-c positions within the sub-sequence of length N. Then, for any integer N (where N runs from 1 to infinity), all K^c conditional probabilities of c particular specified values at the c consequent positions (given that g particular specified antecedent values have appeared at the g antecedent positions), are all equal to $\frac{1}{\kappa^{c}}$ (within an acceptably small error $\epsilon \ge 0$). Note that if g is strictly less than N-c, then there are N-g-c positions within the sequence which are part of neither the antecedent nor consequent part of the prediction. That is, they are "don't care" positions. For a given N, there are multinomial coefficient $\binom{N}{g \ c \ N-g-c}$ ways of picking the g given antecedent positions and c consequent positions out of the subsequence of length N. And then, after g such given antecedent positions are picked, there are then K^g particular assignments of values at the g antecedent positions. After those choices are made, there are then K^c particular assignments of values at the c consequent positions. The conditional probabilities of the K^C particular assignments of values that should be equal.

The above definition captures the intuitive notion based on conditional probabilities; however, it is particularly onerous combinatorially. The following simpler definition avoids conditional probabilities and is sufficient to guarantee satisfaction of the above conditional probability test. For any integer N (where N runs from 1 to infinity), the probabilities of each of the K^N possible sub-sequences of length N are all equal to $\frac{1}{K^N}$ (within an acceptably small error $\epsilon \ge 0$).

No finite sequence can satisfy the above test. However, if N is then limited to some finite fixed integer N_{max} , then "only" K^{Nmax} probabilities must be estimated when N = N_{max} . Moreover, these K^{Nmax} separate probabilities can be conveniently summarized into a scalar quantity by using the concept of entropy for this set of events and probabilities. The entropy (which is measured in bits) is maximal when the probabilities of all the possible events are equal. Moreover, when K = 2, this maximal value of entropy happens to equal the length N (in bits) of the sequence. The entropy E_h for the set of K^h probabilities for the K^h possible subsequences of length h, equals

$$\mathrm{E}_{h} = - \sum \ \mathrm{P}_{hj} \log_2 \mathrm{P}_{hj} \; . \label{eq:eq:ehermic_hamiltonian}$$

Here j ranges over the K^h possible sub-sequences of length h. By convention, $log_2 0$ is 0.

If K = 2, then this sum attains its maximum value of h precisely when the probabilities of all the K^h possible sub-sequences of length h are equal to $\frac{1}{2^{h}}$.

As h runs from 1 to N_{max} , it is convenient to further summarize the N_{max} separate scalar values of entropy into one single scalar value, namely, E_{total} as follows:

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

When Etotal attains the maximal value of

$$\sum_{h=1}^{N_{max}} h = N_{max}(N_{max} - 1),$$

then the sequence may be viewed as random.

To illustrate for K = 2: When h is 1, there are only two probabilities to consider, namely, the probability P_{10} of occurrence of zeroes and the probability P_{11} of occurrence of ones in the entire random sequence. These two probabilities are what are used in Knuth's equidistribution (frequency) test. For the worst randomizer (say one that always emits 1), P_{10} is 0.0 and P_{11} is 1.0 and the entropy is 0.0 bits. For a better randomizer, both P_{10} and P_{11} would equal approximately 0.5 and the entropy would approximately equal the maximal value of 1.0 bits. However, a defective randomizer such as one that emits

0101010101...

has entropy of 1.0 bits when only the two "singlet"

probabilities associated with h = 1 are considered. However, when h = 2, the "pair" probabilities of the four possible pairs (i.e. 00, 01, 10, and 11) are not all equal to $\frac{1}{4}$. The two pairs 00 and 11 do not appear at all in the output of this defective randomizer. The two pairs 01 and 10 appear with probability $\frac{1}{2}$. Thus, the entropy for h = 2 for this defective randomizer is only 1.0, instead of the maximal value of 2.0 bits possible.

3. TYPES OF PSEUDO-RANDOM NUMBER GENERATORS

The common types of randomizers start with one or more seeds [1]. Multiplicative congruential randomizers start with a seed value x_0 and then produce subsequent elements of the sequence recursively as follows:

$$x_i = (a x_{i-1} + c) \mod M$$
,

where a is the multiplier, c is the additive constant, and M is the modulus. Park and Miller [19] describe the especially simple and popular randomizer

$$x_i = 7^5 x_{i-1} \mod [2^{31} - 1].$$

which provides especially good randomness by many tests for the low-order bits.

The popular URN08 randomizer came from IBM in 1970 and is widely known as "RANDU". IBM's RANDU is the multiplicative congruential randomizer

$$x_i = 65539 x_{i-1} \mod 2^{31}$$

Shift register randomizers start with a seed value x_0 and then produce subsequent elements of the sequence recursively in a shift register. In the popular SR[3,28,31] shift register randomizer (called "SHIFT REGISTER" herein), the numbers 3 and 28 specify the amount of shifting (end off, with zero fill) to the right or left (respectively) in a 31-bit shift register. With XOR being the exclusive-or operation,

Shuffling randomizers call on one or more other randomizers to shuffle numbers to produce random numbers. One two-sequence shuffling randomizer (called SHUFFLE herein) uses the Park-Miller randomizer described above to produce an initial set of uniformly distributed random numbers between 0.0 and 1.0 and then uses the shift register randomizer SR[3,28,31] to call out particular numbers from this set of numbers, while using additional calls on Park-Miller [19] to replace the numbers called out.

Texas Instruments supplies a randomizer called

RANDOM with its Explorer[™] computers.

Because of the size of tables needed for the subsequence probabilities needed to compute the total entropy, we will focus, for the remainder of this paper, on producing sequences of random binary digits (i.e. K = 2).

4. BACKGROUND ON GENETIC PROGRAMMING PARADIGM

We have recently shown that entire computer programs can be genetically bred to solve problems in a variety of different areas of artificial intelligence, machine learning, and symbolic processing. Specifically, this recently developed genetic programming paradigm has been successfully applied [4, 5, 12] to example problems in several different areas, including

- planning (e.g. navigating an artificial ant along a trail and developing a robotic plan for stacking blocks in to a desired order) [4, 6],
- emergent behavior (e.g. discovering a computer program for locating food, carrying food to the nest, and dropping pheromones, which, when executed by all the ants in an ant colony, produces interesting higher level "emergent" behavior) [10],
- machine learning of functions (e.g. learning the Boolean 11-multiplexer function) [11],
- automatic programming (e.g. solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities) [4],
- discovering inverse kinematic equations (e.g. to move a robot arm to designated target points) [12, 16],
- optimal control (e.g. centering a cart and balancing a broom on a moving cart in minimal time by applying a "bang bang" force to the cart) [13, 14],
- pattern recognition (e.g. translation-invariant onedimensional shape in a linear retina) [4],
- sequence induction (e.g. inducing a recursive procedure for generating sequences such as the Fibonacci and the Hofstadter sequences) [4],
- symbolic "data to function" regression, integration, differentiation, and symbolic solution to general functional equations (including differential equations with initial conditions, and integral equations) [4],
- empirical discovery (e.g. rediscovering Kepler's Third Law, rediscovering the well-known non-linear econometric "exchange equation" MV = PQ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy) [8],

- concept formation and decision tree induction [9],
- finding minimax strategies for games (e.g. differential pursuer-evader games, discrete games in extensive form) by both evolution and co-evolution [7], and
- simultaneous architectural design and training of neural networks [15].

A visualization of the application of the genetic programming paradigm to planning, emergent behavior, machine learning, empirical discovery, inverse kinematics, and game playing can be viewed in the *Artificial Life II Video Proceedings* videotape [16].

In the genetic programming paradigm, the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals used typically includes inputs (sensors) appropriate to the problem domain and various constants. Each function in the function set must be well defined for any combination of elements from the range of every function that it may encounter and every terminal that it may encounter.

The search space is the hyperspace of all possible compositions of functions that can be recursively composed of the available functions and terminals.

The symbolic expressions (S-expressions) of the LISP programming language are an especially convenient way to create and manipulate the compositions of functions and terminals described above. These S-expressions in LISP correspond directly to the "parse tree" that is internally created by most compilers.

The basic genetic operations for the genetic programming paradigm are fitness proportionate reproduction and crossover (recombination). The crossover (recombination) operation is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. The crossover operation creates new offspring S-expressions by exchanging sub-trees (i.e. sublists) between the two parents. Because entire sub-trees are swapped, this crossover operation always produces syntactically and semantically valid LISP S-expressions as offspring regardless of the crossover points. For example, consider the two parental S-expressions:

(OR (NOT D1) (AND D0 D1))

(OR (OR D1 (NOT D0)) (AND (NOT D0) (NOT D1))

These two S-expressions are depicted as rooted, pointlabeled trees with ordered branches in Fig. 1.



Figure 1: Two Parental LISP S-expressions shown as trees with ordered branches.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the second point (out of 6 points of the first parent) is randomly selected as the crossover point for the first parent and that the sixth point (out of 10 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the NOT in the first parent and the AND in the second parent. The two crossover fragments are two sub-trees shown in Figure 2 below:



Figure 2: The Two Crossover Fragments

These two crossover fragments correspond to the bold, underlined sub-expressions (sub-lists) in the two parental LISP S-expressions shown above. The two offspring resulting from crossover are shown in Figure 3 below. Note that the first offspring above is an Sexpression for the even-parity function, namely

(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)).



Figure 3: Offspring Resulting from Crossover

5. BREEDING A RANDOMIZER

Our goal is to genetically breed a computer program to convert a sequence of consecutive integers into a sequence of random binary digits. The input to our randomizer will merely be an argument J running consecutively from 1 to 16384 (214). Randomizers of this type can quickly reconstruct a particular single random number within a sequence without having to reconstruct the entire sequence. This type of randomizer typically has a complex structure because it does not rely on a recursive seed. A randomizer using a recursive seed as its input gains considerable computational leverage from the cascading of its own previous steps. For example, when the third call is made to a recursive randomizer with initial seed x_0 in order to compute x₃, the randomizer is, in effect, computing

 $x_3 = (a[(a \{(a x_0 + c) \mod M\} + c) \mod M] + c) \mod M.$

The first major step in using the genetic programming paradigm is to identify the set of terminals. The terminals in the genetic programming paradigm correspond to the inputs to the computer program being genetically bred. Thus, the terminal set for this problem is the set $T = \{J, \leftarrow\}$, where \leftarrow represents small random integer constants between 0 and 3.

The second major step in using the genetic programming paradigm is to identify a sufficient set of functions for the problem. The protected modulus function MOD% and protected integer quotient function QUOT% uses the "protected" division function %, which returns one when division by zero is attempted, and, otherwise, returns the normal quotient. Since we anticipate creation of a randomizer involving congruential type steps, the function set for this problem

F = {+, -, *, QUOT%, MOD%}.

The third major step in using the genetic programming paradigm is identification of the fitness function for evaluating how good a given computer program is at solving the problem at hand. The raw fitness here is the sum of the entropy measures E_{total} (described earlier) for sub-sequences of length 1 through 7. The maximum raw fitness is therefore 28.

The fourth major step in using the genetic programming paradigm is selecting the values of certain parameters. The most important parameter is population. The population size is 500 here. Each new generation was created from the preceding generation by applying the fitness proportionate reproduction operation to 10% of the population and by applying the crossover operation to 90% of the population (with one parent selected proportionate to fitness). In selecting crossover points, 90% were internal (function) points of the tree and 10% were external (terminal) points of the tree. Mutation was not used. For the practical reason of conserving computer time, the depth of initial random Sexpressions was limited to 4 and the depth of Sexpressions created by crossover was limited to 15.

Finally, the fifth major step in using the genetic programming paradigm is the criterion for terminating a run and accepting a result. We will terminate a given run when either (1) the genetic programming paradigm produces computer program whose entropy of 27.990 or greater, or (2) 51 generations have been run. Double precision arithmetic was used in computing entropy.

Note that the first two of these five major steps in applying the genetic programming paradigm corresponds to the step (performed by the user) of determining the representation scheme in the conventional genetic algorithm operating on character strings (that is, determining the chromosome length, alphabet size, and the mapping between the problem and chromosomes).

In addition, note that the step (performed by the user) of determining the set of primitive functions in the genetic programming paradigm is equivalent to a similar step in other machine learning paradigms. For example, this same determination of primitive functions occurs in the induction of decision trees using ID3 [9] when the user selects the functions that can appear at the internal points of the decision tree. Similarly, this same determination occurs in neural net work when the user selects the external functions that are to be activated by the output of a neural network. The same user determination occurs in other machine learning paradigms (although the name given to this omnipresent determination varies and is often considered by the researcher to be implicit in the statement of his or her problem).

5.1. DESCRIPTION OF ONE RUN

The initial random generation of randomizers was, predictably, highly unfit. In one particular run, 56% of the 500 individuals had entropy 0.000 (usually because the S-expression merely emitted a constant). Another 14% of the population had entropy between 0.001 and 0.080 (out of a possible 28.000 bits). These individuals emitted a constant almost all of the time. Another 24% of the population had entropy between 7.000 and 7.056 bits. These S-expressions scored 7.000 because they mapped the consecutive sequence of integers J into another consecutive sequence.

The best 24 individuals from generation 0 scored between 10.428 and 20.920 bits of entropy. The best single S-expression scored 20.920 bits. It had 63 points, but can be simplified to 13 points, namely,

(+ J (* (* (MOD% J 3) 3) (QUOT% (+ J 1) 4))).

This individual attains its 20.920 bits (out of a possible 28.000) by getting a perfect 1.000 bits out of a possible 1.000 bits for sequences of length 1; getting 1.918 out of 2.000 bits for length 2; getting 2.792 out of 3.000 bits for length 3; and getting 3.268 out of 4.000 bits for length 4. Although this individual does a credible job of randomizing bits when the window is narrow, it does not do as well for the longer sequences. It gets only 3.689 bits out of 5.000 for length 5 (i.e. only 74% of the possible 5.000 bits); 4.002 bits out of 6.000 for length 6 (i.e. only 67% of the possible 6.000 bits); and 4.252 bits out of 7.000 for length 7 (i.e. only 61% of the possible 7.000 bits).

After 2 generations of one typical run, the entropy of the best-of-generation individual improved to 22.126 bits. After 4 generations, the entropy of the best-of-generation individual improved to 26.474 bits. Thereafter, entropy reached and slowly improved within the 27.800 to 27.900 area.

On generation 14, we obtained an individual Sexpression that attained a nearly maximal entropy of 27.996. It had 153 points, but simplifies to

```
(- J (QUOT% (+ (+ (+ J J) J) (* (+ J 2)
J)) (+ (MOD% (* (- 2 1) (QUOT% (QUOT% (+
(* J J) (QUOT% (- (QUOT% (* J (MOD%
(QUOT% J 3) (MOD% J J))) (QUOT% (* 3 2)
(QUOT% 2 1))) (- 3 (QUOT% (+ (* J J) (- 2
1)) 3))) (* 3 (+ (MOD% 1 0) J))) 3) 3))
(+ (- 2 J) 1)) (+ (QUOT% (MOD% J 3) (-
(MOD% 2 0) (MOD% (MOD% 0 J) J))) (- 3
3)))))
```

The simplified version of this individual (with only 41 points) is graphically depicted in Figure 4 below as a rooted, point-labeled tree with ordered branches:



Figure 4: A Graphical Representation of the Genetically Bred Randomizer after Simplification

In scoring 27.996, this randomizer achieved a maximal value of entropy of 1.000, 2.000, 3.000, 4.000, 5.000, and 6.000 bits for sequences of lengths 1, 2, 3, 4, 5, and 6, respectively, and a near-maximal value of 6.996 for the 128 (2^7) possible sequences of length 7.

Note that the progressive change in size and shape of the individuals in the population is a characteristic of the genetic programming paradigm. The size (153 points) and shape of the best scoring individual from generation 14 differs from the size (63 points) and shape of the best scoring individual from generation 0. The size and particular hierarchical structure of the best scoring individual from generation 14 was not specified in advance. Instead, the entire structure evolved as a result of reproduction, crossover, and the relentless pressure of the fitness (i.e. entropy).

5.2. RESULTS OF SEVERAL RUNS

In the previous section we described one particular run of the genetic programming paradigm in which we obtained a randomizer with entropy of 27.996. The genetic programming paradigm (as with genetic algorithms in general) contains probabilistic steps at several different points. As a result, we rarely obtain a solution to a problem in the precise way we anticipate and we rarely obtain the precise same solution twice.





We can measure the number of individuals that need to be processed by a genetic algorithm to produce a desired result (i.e. entropy of 27.990 or better) with a certain probability, say 99%. Suppose, for example, a particular run of a genetic algorithm produces the desired result with only a probability of success p_s after a specified choice (perhaps arbitrary and non-optimal) of number of generations N_{gen} and population of size N. Suppose also that we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon =$ 99%. Then, the number K of independent runs required is

$$K = \frac{\log (1-z)}{\log (1-p_S)} = \frac{\log \varepsilon}{\log (1-p_S)}, \text{ where } \varepsilon = 1-z.$$

For example, we ran 10 runs of this problem with a population size of 500 and 51 generations. The graph in Figure 5, shows that the probability of success p_s of a run is 90% with 15 generations. With this probability of success $p_s = 0.90$ on a particular single run, we need K = 2 independent runs with a population of 500 for 15 generations to assure a 99% probability of solving this problem on at least one run. That is, processing 15,000 individuals is sufficient.

5.3. DISCUSSION

Is the genetically bred computer program a good randomizer? The answer is a decisive "Yes and No."

Table 1, below, compares the shortfall in entropy from the maximal 28.000 bits for the genetically bred randomizer and the five commercial randomizers described earlier.

Randomizer	Entropy Shortfall
Park-Miller	.009
IBM RANDU	.010
SHIFT REGISTER	.010
SHUFFLE	.015
TI RANDOM	.009
Genetic	.004

Table 1: Shortfall in Entropy for the Different Randomizers

As can be seen, the genetically bred randomizer has precisely the characteristic for which it was bred (i.e. high entropy). With respect to that particular measure, it exceeded the performance of the other randomizers.

Table 2: Results of Equidistribution Test

Randomizer	# of 0's	# of 1's	χ ²
Park-Miller	8146	8238	0.52
IBM RANDU	8149	8235	0.45
Shift-Register	8234	8150	0.43
SHUFFLE	8234	8150	0.43
TI RANDOM	8235	8149	0.45
Genetic	8207	8177	0.05

If we apply the equidistribution (frequency) test from Knuth to the genetically bred randomizer and the five commercial randomizers, we get the results shown in Table 2. The genetic randomizer comes much closer to the uniform distribution of 8192 zeroes and 8192 ones in a random sequence of 16,384 binary digits than any of the five commercial randomizers. That is, the frequencies of zeroes and ones for the genetic randomizer are almost perfectly uniform. This is understandable, since the genetic randomizer was bred with the entropy fitness measure explicitly so as make single bit frequencies (and indeed sub-sequence frequencies up to length 7) uniform. The greater the deviation from uniformity, the worse the fitness (entropy).

But, in another sense, the frequency of zeroes and ones in the genetic randomizer leans in the direction of being too uniform. When we compute χ^2 (chi-square), we find that the five commercial randomizers have a χ^2 which, for one degree of freedom, is very close to the 50th percentile. In contrast, the χ^2 of the genetically bred randomizer is in about the 15th percentile. Whether the intentionally created uniform distribution or a less uniform distribution constitutes randomness depends on ones point of view. For example, some state lotteries explicitly enforce maximal uniformity (entropy) in the distribution of winners in preprinted instant lottery game tickets as opposed to "unstructured randomness."

If we apply the gap test from Knuth [17] to the genetic randomizer and the five other randomizers, we get the results shown in Table 3 below. The gap test counts the number of gaps of particular sizes in the sequence of 16,384 binary digits. For example, in the sequence 0010110 of length 7, we have one gap in the zeroes of length zero (i.e. one instance of consecutive zeroes), one gap of length one, and one gap of length two.

Randomizer	χ ² for 0's	χ ² for 1's
Park-Miller	5.52	12.80
IBM RANDU	9.07	13.19
Shift-Register	11.21	4.05
SHUFFLE	9.22	10.72
TI RANDOM	4.50	9.44
Genetic	9.03	2.64

Table 3: Results of the Gap Test

The 50th percentile of the χ^2 distribution for 10 degrees of freedom is at 9.342. The χ^2 for the zeroes for the genetic randomizer is 9.03. This is similar to most of the values for the five other randomizers. However, the χ^2 for the ones for the genetic randomizer is the rather low value of 2.64. That is, the gaps in the ones for the genetic randomizer almost perfectly match the expected number of gaps expected from a random source. That is, the genetic randomizer adheres too closely to the theoretical distribution of gaps. This match is especially strong for gap sizes of seven and below. This is understandable in view of how the genetic randomizer was bred. In contrast, the five other randomizers are "more random" because their performance deviates more sharply from the theoretical distribution of gaps expected from a random source. As Knuth said, "...the observed values are so close to the expected values, we cannot consider the result to be random!"

6. CONCLUSIONS

We demonstrated that it is possible to use the genetic

programming paradigm to breed a computer program to perform the task of converting a sequence of consecutive integers into a sequence of pseudo-random bits with near maximal entropy. The size and shape of the initial, intermediate, and final programs were not specified in advance. Instead, the size, shape, and specific internal steps of the various computer programs emerged from a evolutionary process driven by the selective pressure applied by the fitness (entropy) measure. The better programs produced in the later generations of this evolutionary process were structurally more complex than the earlier programs.

7. REFERENCES

- S. L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*. 32(2). Pages 221-251. June 1990.
- [2] K. A. De Jong. On using genetic algorithms to search program spaces. Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates, Hillsdale, NJ 1987.
- [3] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI 1975.
- [4] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In Proceedings of the 11th International Joint Conference on Artificial Intelligence. San Mateo, CA: Morgan Kaufmann 1989.
- [5] J. R. Koza. Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Stanford University Computer Science Dept. Technical Report STAN-CS-90-1314. June 1990.
- [6] J. R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI*. Washington. November, 1990. IEEE Computer Society Press, Los Alamitos, CA 1990.
- [7] J. R. Koza. Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer, Jean-Arcady and Wilson, Stewart W. From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior. Paris. September, 1990. MIT Press, Cambridge 1991.
- [8] J. R. Koza. A genetic approach to econometric modeling. In Bourgine, Paul and Walliser, Bernard. *Proceedings of the 2nd International*

Conference on Economics and Artificial Intelligence. Pergamon Press 1991.

- [9] J. R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In Schwefel, Hans-Paul and Maenner, Reinhard (editors) *Parallel Problem Solving from Nature*. Springer-Verlag, Berlin, 1991.
- [10] J. R. Koza. Genetic evolution and co-evolution of computer programs. In Farmer, Doyne, Langton, Christopher, Rasmussen, S., and Taylor, C. (editors) Artificial Life II, SFI Studies in the Sciences of Complexity. Volume XI. Addison-Wesley, Redwood City CA 1991.
- [11] J. R. Koza. A hierarchical approach to learning the Boolean multiplexer function. In Rawlins, Gregory (editor). Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems. Bloomington, Indiana. July, 1990. San Mateo, CA Morgan Kaufmann 1991.
- [12] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1991 (forthcoming).
- [13] J. R. Koza and M. A. Keane. Genetic breeding of non-linear optimal control strategies for broom balancing. In *Proceedings of the Ninth International Conference on Analysis and Optimization of Systems. Antibes, June, 1990.* Pages 47-56. Springer-Verlag, Berlin, 1990.
- [14] J. R. Koza and M. A. Keane. Cart centering and broom balancing by genetically breeding populations of control strategy programs. In *Proceedings of International Joint Conference on Neural Networks, Washington, January, 1990.* Volume I. Hillsdale, NJ: Lawrence Erlbaum 1990.
- [15] J. R. Koza and J. P. Rice. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of International Joint Conference on Neural Networks, Seattle, 1991.*
- [16] J. R. Koza and J. P. Rice. A genetic approach to artificial intelligence. In C. G. Langton Artificial Life II Video Proceedings. Addison-Wesley 1991.
- [17] D. E. Knuth. The Art of Computer Programming. Volume 2. Addison-Wesley, Reading, MA, 1981.
- [18] G. Marsaglia. Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci. U.S.A.*, 61, pages 25-28. 1968.
- [19] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Comm. ACM*. 31, pages 1192-1201, 1988.