Cracking and Co-evolving Randomizers

Jan Jannink
Stanford University
Computer Science Dept.
Stanford, CA 94305
e-mail: jan@cs.stanford.edu

Jan Jannink

Although **pseudo-random number generators** or **randomizers** are of great importance in the domain of simulating real world phenomena, it is difficult to construct functions which satisfy the many criteria, such as uniform distribution, which 'good' randomizers possess. It is computationally expensive to perform the statistical analysis required to establish their quality. Moreover, no current method of analysis can guarantee quality, since even the question of what constitutes the set of criteria defining randomness remains open.

This paper discusses two experiments designed to test the applicability of genetic programming to the analysis and the unconstrained construction of randomizers. The first experiment attempts to unravel the structure of a number of generators recommended in the literature by guessing their output, given previous output data. The second examines the possibility that co-evolving populations of programs in a competitive environment can produce randomizers which conform to the criteria without explicitly testing for them. In other words, the requisite properties must emerge from the experiments' nature.

The experiments have a convenient representation as a guessing game, closely resembling the two player penny matching problem, in which each player choses heads or tails, and one player hopes to outwit the other in order to prevent a match between the pennies, while the other tries to guess what the first will to do and force a match.

Such competition between programs in the genetic programming framework should breed functions whose output sequence is difficult to reproduce. Through this simple mechanism we strive for a further result, a functional approach to randomness, rather than its statistical description. This would have the advantage of being similar to the definition as put forth in information theory.

Finally, in addition to introducing novel fitness measurement techniques, these simulations aim to show that the genetic programming paradigm is suitable for building structured models of randomness from limited information, without using the exhaustive traditional tests of randomness.

20.1 Background

The themes which pervade the discussion that follows are randomness, game theory, co-evolution, information theory and competition, with genetic programming as a binding force linking them together.

Random or stochastic phenomena appear commonly in nature, as well as in human activities such as economics, therefore computer modeling and simulation typically require functions producing randomized sequences of numbers.

These so-called pseudo-random sequences present many of the same characteristics as truly random numbers, if their generators are well designed. Among the important criteria for randomness are uniform distribution of values throughout their range, as well as uniformity of k-tuples of the values. This is defined by considering the k-tuples as coordinates in k-dimensional space, and noting the distance between the hyperplanes they form in the k-space [L'Ecuyer 1988]. Another often used term for this property is high entropy, to indicate that there is little clustering of values.

Randomizer-like functions with this property have been produced with genetic programming [Koza 1992], and it is noteworthy that the GP environment used for these experiments contains an implementation of the "minimal standard" generator [Park, Miller 1988].

That randomness is a non-trivial matter is readily apparent from the considerable literature devoted to the question of what constitutes a 'good' or 'bad' generator [L'Ecuyer 1990], and the well documented examples of poor randomizers sold with many commercial computer systems [Park, Miller 1988].

Most tests for randomness require large amounts of computer time, and are sensitive to the initial conditions of the tests [L'Ecuyer 1988]. One goal in the study of randomizers is to reduce the time required to test them. Another goal is to derive a functional description of the randomness they attain, or in other words to give a measure of how difficult it is to differentiate the pseudo-random sequences from truly random numbers.

Game theory provides a possible method of attaining these goals in the simple guise of the two player zero-sum penny matching game which was presented in Von Neumann and Morgenstern's seminal work [Von Neumann 1944]. There is no deterministic strategy for penny matching which will result in success for either player over the long run [Selfridge 1989].

In order for fully randomized behavior to emerge from the penny matching game, an adaptive technique must be devised to take advantage of its format. The competition between the player trying to force a match, and the player trying to avoid it, provides a perfect arena for the use of co-evolution [Hillis 1992].

Indeed, we may find it extremely time-consuming to test whether a sequence is random, but it is relatively easy to check whether another sequence, a guessing strategy developed for the game, matches it closely. This method of testing gives easily calibrated results which depend on the quality of the sequences produced while playing [Angeline, Pollack 1993].

The assumption is that the programs will adapt, and must evolve successfully to match the sequence they are made to reproduce, or else be condemned never to reach their goal. In the latter case we can assume either that the problem is ill defined, or that the tested sequence is simply not amenable to extrapolation (and therefore equivalent in some sense to a random sequence).

The next two sections, **20.2-20.3**, cover the reasons for pursuing this problem. After those, **20.4-20.5** define the methodology of the research, followed by a description of the experiments and their results, **20.6-20.7**, and closing with some thoughts about future work in the area in **20.8**.

Table 20.1
Physical, mathematical and computational phenomena by phase complexity

phase	constant	periodic	*complex*	chaotic	stochastic
Material	molecule	crystal	*liquid*	gas	
Function	linear	repeating	*fractal*	pseudo-random	random
Computation	regular	recursive	*universal*	divergent	

20.2 Motivation

Randomizers are essentially a juggling act between the exigencies of compact representation, rapid evaluation and high entropy. According to Kolmogorov's information theoretic definition of randomness, a random sequence can not be written in a more compact form, such as a program, than the enumeration of its elements [Kolmogorov 1965]. Randomizers absolutely contradict this tenet. Their most desirable property therefore is to produce sequences which for all intents and purposes mimic the characteristics of a random sequence.

In order to situate the randomizer problem in terms of genetic programming, table 20.1 presents an adaptation of a list of phenomena ordered by phase complexity [Rucker 1993] suggesting a representation of chaos in terms of pseudo-randomness. Literature in the field of artificial life generally focuses on the complex phase as the locus of 'interesting' phenomena, such as universal computation, or by analogy genetic programming. Assuming that the table's categorizations are valid, the experiments presented below are unusual in that they attempt to make chaotic phase behavior emerge out of complex phase processes. In anticipation of difficulties related to this difference, we introduce special fitness testing techniques to deal with the problems' atypical data.

Genetic programming has been shown capable of producing functions that have the property of high entropy [Koza 1992]. Unfortunately, the properties of randomizers go beyond simple entropy, and their interactions have not, and may never be, fully described. Therefore a different, non-explicit method is the only hope at the present time for the study of stochastic patterns.

It appears that there have been no prior attempts to induce and test for randomizing behavior through a competitive mechanism, nor any algorithmic methods to infer the quality of a randomizer through a functional description. The focus has been on learning locally successful strategies [Selfridge 1989] and exhaustive testing for randomness [L'Ecuyer 1988]. These experiments aim to bridge that gap.

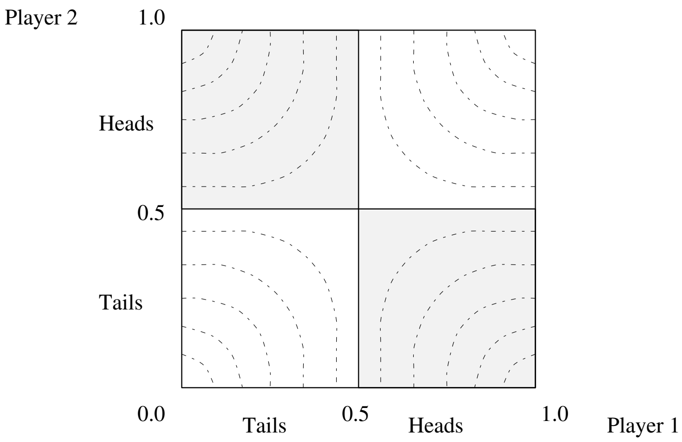

Figure 20.1
Expected gain for players of penny matching game

20.3 Arguments for Success

In order for randomizing behavior to emerge from the experiments there must be reasons why randomness should be a desirable trait to preserve.

20.3.1 Two Player Penny Matching Game

It has been shown that in the penny matching game a random choice of heads or tails is the best strategy for both players, because it minimizes expected loss all the while maximizing expected gain. Penny matching is called the 'natural device' to produce even 50% probabilities [Von Neumann 1944]. Therefore the best programs to solve penny matching should be the ones which exhibit the most random behavior.

Figure 20.1 graphs the expected gain for penny matching players [Selfridge 1989]. The axes give the probabilities of each player choosing heads. The saddle point of the graph is at the center, where both players have an even chance of playing heads. The expected gain is positive in the shaded area for the player trying to prevent the match, negative in the other regions, and vice-versa for the other player.

20.3.2 Uniform Distribution

The most compelling reason to hope for the emergence of a good randomizer out of a penny matching competition is that a distribution of plays which isn't fully uniform is amenable to reverse engineering. If any set of numbers is more likely to appear in the sequences, then it will be targeted and exploited by the guessing programs, and the fitness of the generators would suffer.

20.4 Models

In the descriptions below there are terms which will be used in a particular manner.

player : a program generating or guessing a number sequence

population : a group of players evolving together

program : a parse tree with functions as internal nodes and terminals as leaves

seed : a number which is at the head of a sequence of numbers

20.4.1 Two Player Multi-Penny Matching Game

The idea governing the programs' function is a two person zero-sum game. This game is described as follows:

1. A sequence generator offers up a seed number
2. A guesser program tries to extrapolate the sequence from it
3. The generator's and guesser's sequences are compared
4. The guesser is successful if its extrapolation was accurate

Ordinary penny matching is a similar game in which the number is replaced by a binary (heads or tails) value. The guesser keeps both pennies if its penny matches the generator's, otherwise the generator keeps them.

The only substantive difference in the multi-penny game is that an integer (multi-bit) number is produced, a sequence of which are compared. The information content of the message between the sequence generator and the guesser is therefore much larger. In effect, it corresponds to matching multiple pennies at a time.

The higher level of information exchange provides the potential for better regression, and simplifies to some extent the measurement of the programs' fitness.

20.4.2 Fitness Measure

Two measures of accuracy for the guesser are immediately discernible. The first is the difference between the guess and the target value of each number in the sequence. The second is Hamming distance or the number of bits per guess which differ in the sequences. The resulting value is scaled in a non-linear fashion, to favor more accurate guesses, and to normalize the score to a value between one and zero. The generator's fitness is measured relative to the guesser's, that is: $1 - \mathit{fitness}(guesser)$.

For simplicity, the sequence on which fitness is initially evaluated is a seed and a set of numbers generated immediately following. This appears to be a somewhat weak test because the generator's non-monotonicity implies that a sequence of guesses may be locally accurate, but divergent in the long term.

However, it is vital to see that for fitness to be a useful measure, the test sequences must be presented to programs in a 'natural' order such as: $f(seed), f(f(seed)), f(f(f(seed)))...$ where $f()$ is the randomizer. The choice of a sequence to test is not a trivial matter because given the sequence corresponding to $f(1), f(2), f(3)...$, the problem becomes one of function regression, and guessers very quickly converge to locally perfect solutions, without any ability to generalize.

In order to better judge the value of the results, the best of generation individuals undergo a validation test on a sequence which is not tested for by the GP system during fitness evaluation. If the results of this test are good for some program, its generality is better ensured.

A further refinement to this measure increases progressively the number of fitness cases presented to programs during the course of a GP run. This method is discussed in detail in **20.5.1 Sampling** below.

When a complete fitness measure entails comparing the performance of each program against every other one, as is the norm for co-evolution, it becomes impractical to do so as the problem scales up in size. A partial fitness test in the form of a tournament over the whole population provides a reasonably accurate measure of fitness in a more scalable fashion. This technique is described below in **20.5.2 Fitness Tournament**.

20.4.3 Single Generator

In the simplest version of the multi-penny matching game there is only one sequence generator, and all of the programs attempt to extrapolate its sequence. The generator does not change over time, which may allow the population of guessers to determine its internal structure. This is the the focus of the first experiment outlined in **20.6.1 Tested Randomizers**. The sequence generators chosen for the experiment are widely used, and are presented in the literature as superior in quality.

20.4.4 Separate Generators and Guessers

Two separate populations of evolving programs compete. The fitness of generators is determined by the failure of guessers to come up with the correct sequences. This will evolve parallel sets of programs which for the most part perform the same function, that is, take a seed and generate a sequence of numbers. It may seem to be the most natural way to implement the problem, but it actually gives a considerable fitness advantage to the generators, which are more likely to compete against incorrect guessers, than guessers against similar generators. This argument expresses the fact that in the space of possible programs, most populate the valleys of any problem's fitness landscape. The mechanism below is a compromise that circumvents this weakness.

20.4.5 Sexing Populations

Because the functions of generating sequences and guessing them are so complementary, it is natural to consider merging them into a single population of programs which both generate and guess sequences. However, it may be counterproductive for a program to guess exactly the same sequence as the one it generates. Therefore it is convenient to introduce a tag which differentiates the functionality of generating and guessing. Such a tag could cause its subtrees to execute only if the program is guessing, for example. The complementarity of *generator* and *guesser* functions leads to their description as 'sexes'.

Having a tag in the root node of the program can force the program to act as a generator or guesser exclusively. By allowing fixed and flexible 'sexes' in the same population the highest level of functionality is attained. A more detailed description of the actual construct used in the experiments appears below in **20.6.3 Functions and Terminals**.

20.5 New Techniques

In order to tackle the randomizer problem new approaches to fitness measurement became necessary. Dynamic sampling addresses the issue of overfitting and non-convergence that may occur when too little or conversely too much fitness data is present. The fitness tournament minimizes scalability problems occurring when program fitness must be measured relative to other programs in a population.

20.5.1 Dynamic Sampling

Randomizers are atypical functions in that they are by design non-monotonic. This presents a serious problem to any method of pattern matching attempting to, as it were, perform curve fitting on data produced by a randomizer. If too little data is presented then the pattern matcher is likely to arrive at a locally correct solution only. Moreover, the data appears contradictory and patternless if too much of it is available. To achieve any form of generality there must be a mechanism to allow initially some convergence, and thereafter increase the generality of the solution.

Generally, in genetic programming, the fitness cases correspond to a sampling of the data which comprise a solution to a problem. These are defined at the beginning of the run, and remain unchanged throughout the run. Dynamic sampling proposes to change or add fitness cases throughout the course of the run. This can be on a constant schedule, or a randomized one with a distribution around a given range, or at an increasing rate.

The advantage of this approach for difficult problems is that some initial convergence becomes feasible while the number of fitness cases is small, and that by modifying them or by adding supplementary fitness cases on the fly, a route towards more general solutions becomes a possibility.

20.5.2 Fitness Tournament

This fitness measurement technique overcomes scalability issues when fitness must be calculated relative to the performance of other members of the population, as occurs in problems using co-evolution. It allows the problem size to increase without compromising the accuracy of the fitness measure obtained. It is not to be confused with tournament selection, which finds programs for reproduction and crossover using previously calculated fitness values.

In order to obtain a valid measure of a program's fitness, the whole population must be evaluated in some way. As described above fitness is measured relative to the performance of an opponent. In the case of learning strategies for penny matching when there are only two players, they both evolve towards locally successful strategies for the game, and become caught in chaotic cycles, as one tries to elucidate the other's strategy and still remain unpredictable [Selfridge 1989].

To avoid such cyclic behavior, tournaments between all the players determine an overall best fitness. In the first round all programs compete, and all subsequent rounds consist of smaller tournaments on the winning and losing halves of the previous round. After the tournaments are completed the most successful and weakest strategies have sorted themselves out.

This method's drawback is that there may be classes of strategies which are approximately equally fit, in that they perform well against a similar number of strategies, but never the same ones. This will occur, as a rule, because there is at best a partial ordering of game strategies. By playing multiple tournaments the likelihood of this drawback becoming significant is reduced.

Since every program participates in the same number of tournaments, it is still perfectly acceptable to sort the population by fitness, which remedies the above weakness by retrieving programs which lost early on in the tournament, but performed well otherwise.

The fitness tournament retains most of the accuracy of a complete fitness test with added scalability.

20.6 Experiments

This section describes some implementation details of the experiments, starting with the randomizers examined in the first experiment. The functions, terminals and other parameters constituting the runs are also covered.

20.6.1 Tested Randomizers

The first experiment involves a single randomizer and a population of programs attempting to reproduce its sequence. In order to tune up the system we performed a number of preliminary tests on the trivial randomizing function: $x * 17$ mod 269. The parameters derived as a result of the tuning are listed in table 20.2 below. The randomizers described in this section are promoted in the literature as having good characteristics. Each description comes with a name, to identify it in subsequent discussion.

Two basic types of randomizers are considered here. The multiplicative linear congruential generator introduced by D. H. Lehmer is probably the most widely used method for obtaining pseudo random sequences.

The trivial randomizer above belongs to this category, as well as the Park-Miller randomizer `(r0)` which corresponds to: $(x * 7^5)$ mod $(2^{31} - 1)$. Its main advantage is that it is very easy to start up, since only a single seed value (x) is necessary, and it can be efficiently implemented on small computers. However, its period, the number of numbers generated before its sequence repeats is limited by the size of the modulus (the last number in the equations above).

A few further generators derived from the above model are included in the tests as well. First among these, integer division by four on the Park-Miller randomizer produces a sequence minus two low order bits `(r1)`, which are considered less random than the others. Others recommended in [L'Ecuyer 1988] combine the input of two or three generators as shown in the pseudo-code below.

Figure 20.2
Randomizer with two seeds: `(r2)`

```
seed1 = seed1 * 40014 mod 2147483563
seed2 = seed2 * 40692 mod 2147483399
output = seed1 - seed2
if (output < 1)
  output = output + 2147483562
```

Figure 20.3
Randomizer with three seeds: (r3)

```
seed1 = seed1 * 157 mod 32363
seed2 = seed2 * 146 mod 31727
seed3 = seed3 * 142 mod 31657
output = seed1 - seed2 + seed3
if (seed1 - 705 > seed2)
  output = output - 32362
if (output < 1)
  output = output + 32362
```

The above have a longer period than simple generators of the same size while still being quick to initialize. Also, unlike the simple generators they do not exhibit a lattice structure when consecutive outputs from the randomizer are taken as Cartesian coordinates and plotted [L'Ecuyer 1988].

Lagged Fibonacci generators form the second major group of randomizers. These have the advantage of having long period lengths, but require more initialization. An example of this type is a randomizer proposed by G. Marsaglia with 97 seed values and essentially calculates: $x_n = (x_{n-97} - x_{n-33})$ mod 2^{24} (r4).

Finally, an extended version of the Fibonacci generator with 24 seed values calculating: $x_n = (x_{n-24} - x_{n-9} + c)$ mod 2^{24}, where c represents a 'carry' value set to 1 if in the previous iteration $x_{n-24} < x_{n-9}$ and 0 otherwise (r5). It has a period length of approximately 10^{170}, within which it is simple to generate long disjoint sequences, and would therefore appear to be ideal for heavy users of random numbers [James 1990].

There are other types of randomizers, which do not have as strong a claim to success, a number of which are described and analyzed in [Knuth 1969].

20.6.2 Tableau

In keeping with the format presented in [Koza 1992], table 20.2 gives the initial parameters of the runs. As many results are reported, mnemonic symbols are included to simplify the description of the experimental conditions. The $\bullet$ symbol indicates a default value of the sequence guessing experiment, whereas a $\lhd$ or $\rhd$ refers to a co-evolution experiment. Others appear in the figures or in the text below.

20.6.3 Functions and Terminals

The terminal set is kept as simple as possible. It allows for a variable x (in two tests a second variable y is also used), and the set of random integer constants in the range $[0 - 255]$. The variable x is the output produced by the randomizer in its previous iteration (y is from the iteration prior to x). The purpose of the integer size restriction is twofold. First, it

Table 20.2
Genetic programming parameters associated with a mnemonic

Objective:	•	deduce structure of various randomizers
	◁	co-evolve population(s) of generators and guessers
Terminal set:	•◁	`X`, `random-constant`
	two	`X`, `Y`, `random-constant`
	old	..., `Y`, `random-constant` (`Y` is the older of two prior sequence outputs)
Function set:	•◁	+, −, *, `MOD`, `NOT`, `AND`, `OR`, `XOR`
	▷	+, −, *, `MOD`, `NOT`, `AND`, `OR`, `XOR`, `IFGEN`
	mem	+, −, *, `MOD`, `NOT`, `AND`, `OR`, `XOR`, >>, <<, `READ`, `WRITE`
Fitness cases:	•	15 output pairs from successive randomizer iterations
	chg	15 output pairs changing dynamically, with an average rate of change set to once per 14 generations
	frq	15 output pairs changing dynamically, with an average rate of change increasing from once per 12 generations to once every generation
	inc	15-100 output pairs; new pairs added dynamically with an increasing average rate of change
Validation:	•	10 output pairs from successive randomizer iterations
Fitness score:	•◁	sum over all fitness cases of the difference between randomizer output and tested individuals' output
	bit	sum over all fitness cases of hammimg distance between randomizer output and tested individuals' output
Parameters:	•	1 population of 2048 sequence guessers, 100 generations, 25 runs, tournament selection size 6, crossover 83.2%, reproduction 16.6%
	◁	2 separate populations, 512 guessers, 512 generators, 200 generations, 50 runs, tournament selection size 5, crossover 83.2%, reproduction 16.6%
	▷	1 combined population of 1024 guesser/generators, 200 generations, 50 runs, tournament selection size 5, crossover 83.2%, reproduction 16.6%
Termination predicate:	•◁	reach final generation of run

prevents an explosion in the size of the numbers used in the function, second, it fits in with the instruction set which builds up large numbers more simply than it reduces them.

One function set exists in two versions, the second differing only in the presence of the `IFGEN` 'macro' function which executes its left subtree if it is a *generator*, and its right subtree if it is a *guesser*. The first is used for simulations involving a population of separate program types only, while the second is appropriate for a co-evolving population combining both program types in the same population.

The `MOD` operator is a protected modulo, which returns the first argument if the second is null. The `DIV` operator was left out of the function sets, although it could arguably be included.

A few further functions were added to one set of runs. The first two are functions performing an arithmetic shift to the binary values of the data. The first argument for these functions is the one to be shifted, the second the number of bits it must shift. Here, assuming 32 bit sized data, the second value is restricted to the range $[0-31]$.

A second pair of additional functions are `READ` and `WRITE`, as defined in [Teller 1994], which access an array of 16 values of the same type as the function arguments. The address argument of both functions is restricted to the range $[0-15]$. Before each fitness evaluation in runs using these operators, the array is initialized so that each array element contains a value equal to its index plus one. This provides a well defined set of initial data to the program, and also facilitates the possible use of the array for indirect indexing. A detailed discussion of these operators is not within the scope of this paper and appears in chapter 9.

All functions in these experiments take two arguments except for `NOT` and `READ` which are unary.

20.6.4 GP Shell Modifications

All results described below were derived using the SGPC system [Tackett 1993], compiled with minimal modifications, most notably on its handling of random number generation, running on SUN sparc10 workstations.

To achieve co-evolution of populations several coding changes were necessary. The new `evaluate_fitness_of_populations` function goes through the population matching up pairs of programs in multiple tournaments that simultaneously build an ordering of the population by number of wins.

Also, fitness-cases are handled in a slightly different fashion. The functionality of `define_fitness_cases` is split into two distinct phases. Its new namesake just creates the data structure to contain the fitness values, and sets only a few of them. The function `evaluate_fitness_of_populations` is now also involved in the creation of new fitness cases. Its added functionality is applied during runs, after the generation of new programs is complete.

20.7 Results

The results of the experiments were sensitive to the settings of the global variables controlling the behavior of the GP system. The parameters indicated under the heading of the same name in the tableau were chosen heuristically, on the basis of tests using the trivial randomizer described above. The fittest programs produced by the tests reported here are on the order of 200 lines of code, even when simplified, therefore none are reproduced here.

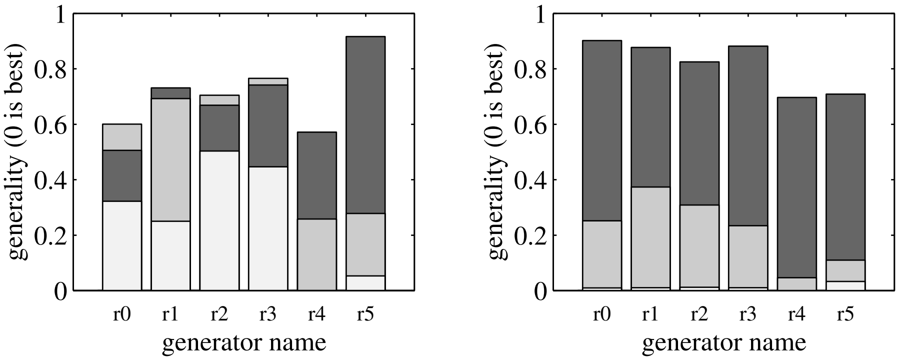

Figure 20.4
Comparison, by validation then fitness, of six sequence guessers by relative improvement during the tests, using as a benchmark value the average score of the fittest generation 0 individuals over all runs. Light shading represents the best individual of the test, dark the best individual of the first generation, and medium the average of the best found in each run.

The following discussion attempts to make up for this deficiency by being as informative and descriptive as possible.

Figure 20.4.a and figure 20.4.b plot values taken from 25 runs, which we call a test, for simplicity, on each of the six randomizers presented above. Referring to the tableau's nomenclature, the default settings indicated by • are active for these tests. All tests are initially identical except for the randomizer producing the sequences the population must reproduce.

From the raw output data, we extract information on the best individuals of each run, and process that down into eight essential values. These values, calculated at the end of each randomizer's test are as follows:

1. average validation score of fittest generation 0 individuals in each run
2. best validation score of fittest generation 0 individuals in each run
3. average validation score of fittest individuals of each run
4. best validation score of the fittest individuals during entire test
5. average fitness score of fittest generation 0 individuals in each run
6. best fitness score of any generation 0 individual
7. average fitness score of fittest individuals of each run
8. best fitness score during entire test

Note that the value in 1. above, typically is not the validation score of the fittest individual of the test in the first generation, but is that of one of the other fittest individuals of the runs.

Fitness and validation scores are not necessarily tightly coupled, as is readily observed in the above charts.

Figure 20.4.a, figure 20.4.b and figure 20.5 measure generality, in other words, the extent to which improvement in fitness, as tested through the fitness cases, reflects fitness changes in an untested data set. Generality is a crucial benchmark in experiments such as these where only a small fraction of the possible fitness cases are used. GP's natural convergence to solutions ensured that performance relative to the tested fitness cases improved by orders of magnitude in each experiment pictured here, a phenomenon which clearly did not always occur with the validation cases.

The plot is normalized so the average validation score of the first generation is equal to 1.0 for each randomizer. The assumption made here is that this score is rather uniformly mediocre, for any simulation. All improvement that GP achieves is in reference to this benchmark value. Direct comparison between randomizers, using this scheme, is prone to some inaccuracy, due to their variation in range, but it is adequate for our purposes.

Each bar in the graph is composed of two or three shaded regions. Those in figure 20.4.a and figure 20.5, correspond to the ratios of the second, third and fourth values from the above list, to the first list value. Figure 20.4.b, on the other hand, displays ratios of the sixth, seventh and eighth list items to the fifth. These ratios are shaded dark, medium and light respectively. In bars having only two shadings, such as `r4`, the missing value was too low to be displayed in the figure.

The equation below calculates the generality value x of the individual, among the fittest of the test, which had the lowest validation fitness score, which appear as the light bars in figure 20.4.a. Here n refers to the number of runs in the test, r a single complete run, $gen0(r)$ the first generation of a run, and $validation(r)$, the validation score of the fittest individual of r. Other generality values are derived in analogous fashion.

$$x = \frac{\min(validation(r_i))}{\sum_{i \in n} validation(gen0(r_i))/n}$$

As is plain from figure 20.4, the randomizers above are extremely diverse in their ability to withstand the test of genetic programming. Although in every case the validation fitness score improved beyond the tested fitness of the fittest individuals of generation 0, indicating a basic level of generality, the actual amount of change served as a good indicator of the randomizers' performance. The lagged Fibonacci randomizers in particular are weak. The normalized score of the fittest program up against `r4` is a vanishing 0.00004. In other words the program had a very high level of generality, since it not only solved the fitness cases accurately, but also passed the validation test with flying colors. The scale of figure 20.4.a

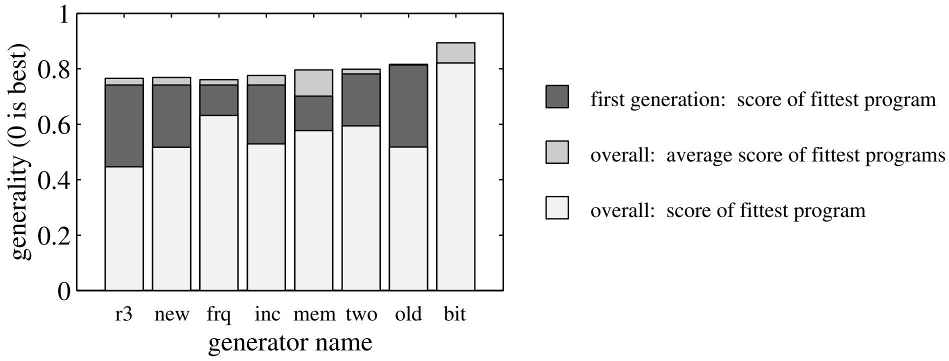

Figure 20.5
Comparison of eight versions of the same sequence guesser, differing in the details of fitness measurement, function set, and terminal set.

is such that the bar corresponding to this result is not visible. Tests with both the defaults and the *inc* sampling method (not shown here) accomplished this validation result, on runs with different starting conditions. This result may actually be due to the initial state of the seeds, to which these randomizers are highly sensitive.

The randomizer with the best performance overall is `r3`. We investigated whether better results are possible for this randomizer and collect the results in figure 20.5 below.

The different tests on `r3` are as follows:

1. initial conditions as in figure 20.4; tableau default: •
2. changing fitness cases with a set average rate; *chg*
3. changing fitness cases with increasing frequency; *frq*
4. increasing the number of fitness cases; *inc*
5. extended function set using a memory array; *mem*
6. two previous sequence outputs as terminals; *two*
7. only the older of two outputs as a terminal; *old*
8. the bitwise hamming distance of the sequences; *bit*

The values graphed for *chg*, are the best of a set of ten tests not shown here, which evaluated changing fitness cases at a fixed rate of once every 6, 8, 10, 12, 14 generations, as well as changing them at a variable rate averaging once in 6, 8, 10, 12 or 14 generations. A slight improvement in average validation score was achieved over the default, at the expense of a less fit overall best individual.

Increasing the frequency of fitness case change, as in *frq*, appears uninteresting from the perspective of absolute results, but there is a mitigating factor. More of the individuals with good validation scores occurred in later generations of the runs, so while their scores were not exceptional, there was more continued progress.

When increasing the number of fitness cases *inc*, not only did overall best scores remain close to the default runs, but the fit individuals continued to evolve longer. The best individual appeared in generation 94, and maintained its high validation score, whereas in the default test the population overfitted to the fitness cases, and the best individual which appeared in generation 20 disappeared immediately, and nothing approaching its score ever returned. Tests on the other randomizers, not included here, provide more impressive evidence of the power of this method, as they all produced better validation scores than any other technique.

Of all the tests, the one using the indexed array memory model *mem*, achieved the best average validation score, although the hoped for score improvements were not. We earlier reported excellent results for this model, and soon discovered that it was the result of a fascinating bug, which in itself opens up vast areas of potential discoveries in genetic programming. Inadvertently, the memory array was not correctly initialized except at the start of each generation. As a result, information was continuously added to the array in the fitness evaluations of the entire population. This property of 'sharing' information, as it were, allowed the population as a whole to evolve towards solutions in a more concerted way, and allowed its best individuals to go beyond what was achieved by any other method. The implications of this bug appear to be countless in the areas of models of communication, cooperation, etc., and offer another direction to explore co-evolution.

It is a disappointment that the test with two terminals, consisting of the two previous outputs of the randomizer *two*, did not result in an improvement. In absolute terms, the best validation score was more than double that of the default test. It appears that the two randomized values are sufficiently uncorrelated to impede the evolution of fit individuals. In this test the best individuals, without exception, made use of only a single terminal, and did not survive long.

Likewise, the test using one terminal which corresponded to the older of the two most recent outputs *old*, did not produce any notable results. Its best individuals achieved a score approximately 40% worse, in absolute terms, than the corresponding test using the more recent output of the randomizer. This result is interesting however, in that it gives an indication of how much information of the randomizer's outputs is retained in the subsequent outputs.

Using hamming distance as a fitness test, *bit*, proved unfruitful, as there were no improvements to the best individual, and only minimal changes in average performance. The

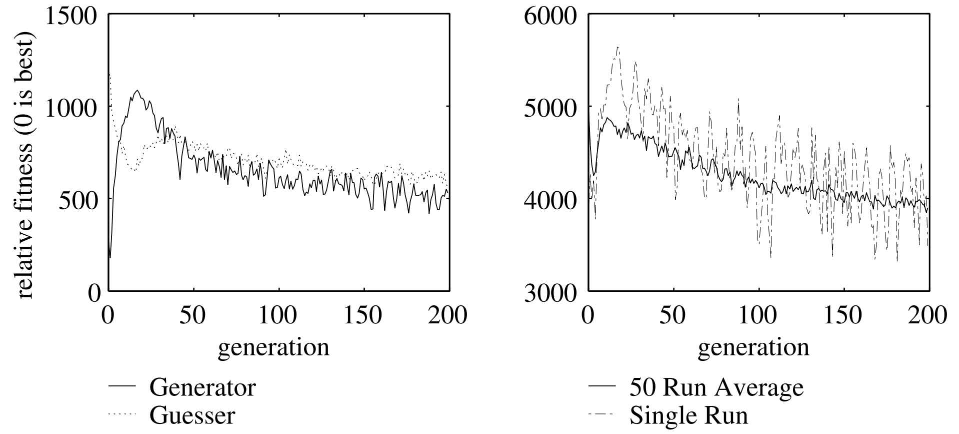

Figure 20.6
Average fitness results over 50 runs of 200 generations each, with co-evolving populations, the first with separate generating and guessing populations, the second with a combined generator/guesser population, showing for contrast an example of a typical single run.

best individual of the test actually appeared in generation 0, which is why in figure 20.5 the dark shading is obscured.

Figure 20.6.a depicts the evolution of fitness for separate populations of guessers and generators (◁ in table20.2), while figure 20.6.b shows the performance of a population of combined guessers and generators (▷). The graphed data corresponds to the average over all runs of the performance of the fittest individual of each run at each generation. As fitness scores in the populations are the result of an internal competition with equal numbers of winners and losers, average fitness of the populations is constant throughout the runs.

In both tests we see an early worsening of fitness. At this stage the guessers learn how to decode the generators' sequences. A plateau is reached when generators begin to modify their techniques. The lines in figure 20.6.a, show how the coupling of generator and guesser result in this pattern. Once this plateau is reached, both populations become fitter in tandem, which is surprising until one refers back to figure 20.1, and realizes that the populations are seeking out the point of maximum gain in their competitions, which for both is the saddle point of the graph.

The bold line in figure 20.6.b demonstrates that the average fitness of these fittest programs follows a pattern of steady decrease, whereas the dashed line indicates the variability of fitness in a single run. The comparison of this data provides a further clue as to what is taking place during the course of a run.

This evidence indicates a phenomenon of discovery by the generators of new methods of encoding data, followed by the corresponding decipherment of this strategy by guessers in succeeding generations. Overall, each new discovery in sequence generation improves the best of generation fitness, although this is masked by the variability of the individual runs.

This tendency for the sequence generation phase to contribute more highly to fitness is borne out by results from runs with separate generators and guessers. When the populations are separate, their coupling is weaker, and there is much greater variability within runs. Therefore a larger population size is necessary to achieve similar results.

20.8 Conclusion

Program evolution simulations establish the counter-intuitive result that it is feasible, up to a certain point, to build increasingly accurate representations of pseudo-random number generators, and thereby obtain a relative measure of their quality.

The mechanism of competition is used to co-evolve populations of programs, which by playing a simple game strive to attain a functional definition of randomness. These experiments show that genetic programming is able to build structured models of randomness from limited information, without using exhaustive traditional tests of randomness.

Fitness measurement techniques, such as progressively increasing the number of fitness cases, promote continued evolution by reducing overfitting, and enable the exploration of problems with extremely noisy data, by initially allowing patterns to be recognized within a minimal set of fitness cases.

Further testing of co-evolved randomizers will show how well they measure up to hand-crafted ones, when subjected to statistical analysis. The concept of a shared memory pool for a population is fascinating for its implications regarding population-wide evolution, and merits its own study.

It is worthwhile to note in passing that Kolmogorov considered, in 1965 already, the concept of algorithmic quantification of the density of information in genetic material. Additional relevant material on randomness and complexity of computer programs, and the mathematical meaning of life is available in [Chaitin 1987].

Acknowledgements

Thanks are due to Patrik D'haeseleer and Jason Bluming for late night inspiration and humor while developing this project. Thanks also to professor Bala Ramesh at the Naval Postgraduate School in Monterey, for providing those most valuable of resources, time and processor cycles. The use of Walter Tackett's SGPC source code is also much appreciated.

Bibliography

Angeline, P. J. (1993) and J. B. Pollack, "Competitive Environments Evolve Better Solutions for Complex Tasks," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest Ed. San Mateo, CA: Morgan Kaufmann.

Hillis, D. (1992) "Co-evolving Parasites Improves Simulated Evolution as an Optimization Procedure," in *Artificial Life II* , C. Langton, C. Taylor, J. Farmer and S. Rasmussen, Ed. Reading, MA: Addison-Wesley Publishing Co.

Chaitin, G. J. (1987) *Information, Randomness & Incompleteness*. Singapore: World Press.

James, F. (1990) "A Review of Pseudorandom Number Generators," in *Computer Physics Communications, vol. 60*, Amsterdam: North-Holland Publishing Co.

Knuth, D. E. (1969) *The Art of Computer Programming, Vol. 2 / Seminumerical Algorithms*. Reading, MA: Addison-Wesely Publishing Co.

Kolmogorov, A. (1965) "Three Approaches to the Quantitative Definition of Information," in *Problems of Information Transmission, vol. 1 no. 1*, New York, NY: Faraday Press.

Koza, J. R. (1992) *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

L'Ecuyer, P. (1988) "Efficient and Portable Combined Random Number Generators," in *Communications of the ACM, vol. 31, no. 6*, New York, NY: ACM press.

L'Ecuyer, P. (1990) "Random Numbers for Simulation," in *Communications of the ACM, vol. 33, no. 10*, New York, NY: ACM press.

Park, S. K. (1988) and K. W. Miller, "Random Number Generators: Good Ones are Hard to Find," in *Communications of the ACM, vol. 31, no. 10*, New York, NY: ACM press.

Rucker, R. (1993) *Presentation on visualizing complexity and 'Bug Land' software, Feb. 2, 1993*, Stanford, CA: Stanford University.

Selfridge, O. G. (1989) "Adaptive Strategies of Learning: A Study of Two-person Zero-sum Competition," in *Proceedings of the 6th International Workshop on Machine Learning 1989*, A. M. Segre Ed. San Mateo, CA: Morgan Kaufmann.

Tackett, W. A. (1993) *SGPC source code*, available via anonymous ftp: `ftp.cc.utexas.edu` under directory `/pub/genetic-programming/code`.

Teller, A. (1994), "The Evolution of Mental Models," in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA: MIT Press.

Von Neumann, J. (1944) and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton, NJ: Princeton U. Press.