Random Generator Quality and GP Performance

Mark M. Meysenburg
Department of Computer Science
Doane College
Crete, NE 68333
mmeysenburg@doane.edu
(402) 826-8267

James A. Foster
Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow, ID 83844
foster@cs.uidaho.edu
(208) 885-7062

Abstract

In previous studies, the authors found that pseudo-random number generator (PRNG) quality had little effect on the performance of a simple genetic algorithm (GA). This paper extends our work to the area of genetic programming (GP). We examine the effect of PRNG quality on the performance of GP techniques. We detail a set of PRNGs which generate random numbers through various techniques, and a method for evaluating the quality of these PRNGs. We explain the application of detailed statistical analysis to the results of more than 50,000 individual GP runs, over a set of four GP test problems. We found no evidence to support the notion that higher quality PRNGs caused improved GP performance.

1 INTRODUCTION

Genetic programming (GP) techniques rely heavily on the use of pseudo-random number generators (PRNGs). From initial population generation, through selection, crossover, and mutation, the use of randomness is pervasive through GP algorithms. Therefore, it is reasonable to wonder how PRNG quality effects GP performance. Previously, Daida, et al. have shown that PRNG choice can effect the solution quality of GP algorithms, albeit in non-intuitive ways. In one case [Daida et al., 1997], they found that a PRNG of lesser quality caused better GP performance than a higher quality PRNG. In another case [Daida et al., 1999], they showed that what appeared to be a large performance gain caused by PRNG selection was in fact not a statistically significant gain. Both of these studies focused on the wall-following robot problem, as described by Koza [Koza, 1992a]. In this study, we examine more PRNGs and different GP test problems, to determine if general PRNG quality effects GP performance.

In our previous work with genetic algorithms (GAs) [Meysenburg, 1997, Meysenburg and Foster, 1997], we found no evidence that improved PRNG quality led to improved GA performance. In this study we applied similar techniques to examine the effect of PRNG quality on the performance of GP algorithms. As for our GA work, we found no evidence that improved PRNG quality led to improved GP performance.

2 TOOLS

We used a modular, object-oriented system consisting of the base GP objects, GP test problems, PRNGs, and a test suite to evaluate the PRNGs.

We used version 1.0 of Kokkonen's `gpjpp` package [Kokkonen, 1997], a Java GP system based on the `gpc++` GP implementation. Like `gpc++`, `gpjpp` supports automatically defined functions (ADFs), tournament and fitness-proportionate selection, demetic grouping, steady state populations, subtree crossover, and swap and shrink mutation. In addition, `gpjpp` produces pictures of expression trees in GIF-file format, implements diversity checking in a relatively efficient manner, implements Koza's greedy over-selection scheme, and allows explicit complexity limits to be set.

We performed experiments using several different Java-coded PRNGs that represent common algorithms for generating pseudo-random numbers. The categories of PRNGs, and the PRNGs themselves, are listed below.

We used two linear congruential PRNGs: RAND and PM. RAND is the `java.util.Random` PRNG, which uses the linear congruential algorithm described in Knuth's Volume 2, section 3.2.1 [Knuth, 1997]. PM is a Java version of the Park and Miller minimal standard linear congruential PRNG. PM is based on the C implementation in Press, et al. [Press et al., 1992].

We used MOTHER, a multiply with carry generator termed "the mother of all PRNGs." MOTHER is described by Marsaglia [Marsaglia, 1994].

We used an additive and a subtractive PRNG, ADD and SUB. ADD is a version of the additive generator described in section 3.2.2 of Knuth's Volume 2 [Knuth, 1997]. SUB is a Java version of the Knuth subtractive generator, based on the C implementation in Press, et al. [Press et al., 1992].

We used three compound, or shuffled, PRNGs: SHSUB, SHPM, and SHLEC. SHSUB is a version of SUB, shuffled using RAND, according to the shuffling algorithm described by Knuth [Knuth, 1997]. SHPM is a shuffled version of PM, based on the C implementation from Press, et al. [Press et al., 1992]. SHLEC is a Java version of the shuffled L'Ecuyer generator, based on the C implementation in Press, et al. [Press et al., 1992].

We used two feedback shift register generators, FSR and TGFSR. FSR is a standard feedback shift register generator of the type described by Schneier [Schneier, 1994]. TGFSR is a twisted generalized feedback shift register generator, described by Matsumoto and Kurita [Matsumoto and Kurita, 1992].

Finally, we used TAUSS, a Combined Tausworthe generator, described by Tezuka and L'Ecuyer [Tezuka and L'Ecuyer, 1991].

We used Marsaglia's Diehard suite of tests, available in executable form on the Internet [Marsaglia, 1993, Marsaglia, 1998], to evaluate the overall quality of our PRNGs. The Diehard suite contains nineteen tests, which operate on large (approximately 10 MB) binary files created by the PRNG in question. The Diehard suite includes several tests similar to those presented by Knuth [Knuth, 1997], such as the Birthday Spacings Test and the Runs Test. In addition, the Diehard suite includes several tests that perform in-depth binary-level examinations of the sequences produced by the PRNG in question. So, the Diehard suite seemed to be a more stringent set of tests than the classic Knuth PRNG evaluation algorithms. The Diehard tests are more fully described in the first author's thesis [Meysenburg, 1997].

We used the four test problems supplied with the `gpjpp` package as our GP test suite. The problems are summarized below. More details regarding test suite parameters can be found at the first author's web cite, `http://www.doane.edu/crete/academic/science/ist/mark.htm`.

We used an artificial ant problem, as described by Koza [Koza, 1992b]. We used the Santa Fe trail for our experiments, Koza's ramped half-and-half method for generating the initial population, and elitist fitness-proportionate (a.k.a. roulette-wheel) selection.

The terminal set for this problem was:

$$T = \{(LEFT), (RIGHT), (MOVE)\}.$$

The function set was:

$$F = \{IFFOODAHEAD, PROG2, PROG3\}.$$

We used a lawnmower problem, as described by Koza [Koza, 1994]. We used a steady-state population, with ramped half-and-half generation of the initial population. We used non-elitist fitness-proportionate selection. With probability $p_n = 0.02$, we allowed the creation of a new individual while evolving the next generation. We used shrink mutation and swap mutation for this problem. Shrink mutation finds a random function gene in a random branch and then replaces that gene by one of its immediate children. Swap mutation finds a random function gene in a random branch and changes the node type of that gene. We used $p_m = 0.02$ for both types of mutation.

The terminal and function sets we used varied slightly from those described by Koza. The terminal sets for the result producing branch, the function $ADF0$, and the function $ADF1$ were the same:

$$T_{rpb} = T_{ADF0} = T_{ADF1} = \{(LEFT), (MOW)\}.$$

The function set for the result producing branch was

$$F_{rpb} = \{ADF0, ADF1, PROGN\}.$$

The function set for $ADF0$ was

$$F_{ADF0} = \{PROGN\}.$$

The function set for $ADF1$ was

$$F_{ADF1} = \{ADF0, PROGN\}.$$

We used a multiplexor problem, also described by Koza [Koza, 1992b]. We used ramped half-and-half generation and non-elitist fitness-proportionate selection.

The terminal set for the result producing branch, the function $ADF0$, and the function $ADF1$ was

$$T_{rpb} = T_{ADF0} = T_{ADF1} = \{A_0, A_1, D_0, D_1, D_2, D_3\},$$

where A_i represented an address input and D_i represented a data input.

The function set for the result producing branch was

$$F_{rpb} = \{AND, OR, NOT, IF, ADF0, ADF1\}.$$

The function set for $ADF0$ was

$$F_{ADF0} = \{AND, OR, NOT, IF\}.$$

The function set for $ADF1$ was

$$F_{ADF1} = \{ADF0, AND, OR, NOT, IF\}.$$

Finally, we used a symbolic regression problem, as described by Koza [Koza, 1992b]. It involves evolving symbolic expressions that approximate the function

$$f(x) = x^4 + x^3 + x^2 + x.$$

We used ramped half-and-half generation and non-elitist fitness-proportionate selection.

The terminal set for this problem was:

$$T = \{X\}.$$

The function set was:

$$F = \{+, -, *, \%\}.$$

3 METHODOLOGY

Since the quality of some PRNGs can be significantly effected by the seed value used to start the sequence, we randomly created a set of thirty-two, sixty-four bit PRNG seed values. We created each seed value by rolling a four-sided die thirty-two times, interpreting each roll as a two-bit value. We then concatenated the bit patterns to form the seed value.

Most of our PRNGs take signed, sixty-four bit integers as seeds; hence our randomly created sixty-four bit binary patterns. However, we derived some of the PRNGs from C implementations; these PRNGs require thirty-two bit seeds. For these generators, we cast the sixty-four bit integers to thirty-two bits. In addition, one generator requires negative thirty-two bit integers; in this case, we cast to thirty-two bits and then flipped the sign bit of positive seeds.

To summarize how a PRNG performed in the Diehard tests, we used a scoring and ranking scheme similar to that in our previous work [Meysenburg and Foster, 1997]. In particular, we ran each PRNG through the test suite, with one run for each of the thirty-two seed values, and then assigned scores and ranks to the PRNGs based on the results of the Diehard tests.

Each of the Diehard tests produced one or more p-values. We categorized the p-values as good, suspect, or rejected, based on Johnson's scheme [Johnson, 1996]. We classified a p-value as rejected if $p \geq 0.998$. We classified a p-value as suspect if $0.95 \leq p < 0.998$. We classified all other p-values as good. We assigned point values to the PRNGs: two points for every reject classification, one point for every suspect classification, and no points for every good classification. Then, we summed these points to produce a final score for each PRNG. Low scores indicate good PRNG quality, whereas high scores indicate poor PRNG quality.

We performed experiments for each of our four GP test suite problems, and for each of our thirty-two seed values. For a particular (test problem, PRNG, seed value) triple, we performed GP runs until a certain number (five for the lawnmower and multiplexor problems, ten for the ant and symbolic regression problems) of solutions meeting the termination criteria were found. A given GP run lasted until a suitable solution was found, or for a maximum of 51 generations.

The `gpjpp` GP system produces volumes of detailed information about each run, including best, worst, and average fitness; best, worst, and average complexity; population diversity, and more. We extracted the generation-by-generation average fitness information from the raw output.

For each test suite problem, we used statistical methods to compare the performance of the GP driven by PRNG a versus the GP driven by PRNG b ($a \neq b$), on a generation-by-generation basis. To do this, we established the following null and research hypotheses.

Null Hypothesis. For this generation, the GP algorithm driven by PRNG a does not perform better than the algorithm driven by PRNG b. In our notation, μ_a and μ_b are the average population fitness for the algorithm driven by PRNG a and PRNG b, and *Opt* is the optimal fitness an individual might achieve:

$$H_0 : |\mu_a - Opt| \leq |\mu_b - Opt|.$$

Research Hypothesis. The inverse of the null hypothesis; in other words, for this generation, the algorithm driven by PRNG a does perform better than the algorithm driven by PRNG b:

$$H_r : |\mu_a - Opt| > |\mu_b - Opt|.$$

For a given generation, we used the fitness of individuals in the population as the measure of GP performance.

We used the Wilcoxson test, described by Green and Margerison [Green and Margerison, 1978], as our statistical measure. This is a method for hypothesis testing using the mean of a random variable with an unknown distribution.

We performed the Wilcoxson test on a generation-by-genration basis, for each (test problem, PRNG a, PRNG b) triple. That is, for each generation n, $1 \leq n \leq 51$, we used the test to compare the average fitnesses in all runs driven by PRNG a with the average fitnesses in all runs driven by PRNG b. We only performed the tests when there were enough data points for it to be meaningful. For example, if PRNG a and PRNG b both drove the GP to valid solutions within ten generations, we performed no tests for generations 11 through 51.

After performing the Wilcoxson tests we tallied the total number of null hypothesis rejections, across all generations, for each (test suite problem, PRNG a, PRNG b) triple. We normalized these tallies to be between zero and one, so that one represented the case where the null hypothesis was rejected in every case. Zero represented the case where there were no null hypothesis rejections.

4 RESULTS

The Diehard scores for our PRNGs are shown in Table 1. As expected, the very common linear congruential PRNGs, PMPM and RAND, scored much worse than the rest of the PRNGs. Given that PRNGs like SHSUB, FSR, and ADD are relatively easy to implement and often faster than linear congruential PRNGs, one wonders why developers continue to include linear congruential PRNGs as part of new language implementations.

However, in our study we wished to determine if better PRNGs make GP techniques perform better. If improved PRNG quality caused improved GP performance, one would expect that the GP would perform better when driven by SHSUB, FSR, or ADD than when driven by PM or RAND. In this study we found no evidence to support this expectation.

Figures 1, 2, 3, and 4 illustrate the tallies of null hypothesis rejections we collected during our statistical analysis.

In the figures, a relatively tall column would mean that there were a large number of null hypothesis rejections for that population size, test suite function, PRNG a, and PRNG b; in other words, there was statistical evidence that the null hypothesis PRNG caused the GP algorithm to perform better than the research hypothesis PRNG.

Further, a line of tall columns parallel to the null hypothesis axis would indicate that the research hypothesis PRNG labeling the line caused the algorithm to perform worse than all other PRNGs in the study. A line of tall columns parallel to the research hypothesis axis would indicate that the null hypothesis PRNG labeling the line caused the algorithm to perform better than all other PRNGs in the study.

PRNG	Rank	Score
SHSUB	1	548
FSR	2	573
ADD	3	577
TGFSR	4	584
MOTHER	5	602
SUB	6	655
SHLEC	7	751
SHPM	8	799
TAUSS	9	935
PM	10	1619
RAND	11	2129

Table 1: PRNG Test Suite Scores

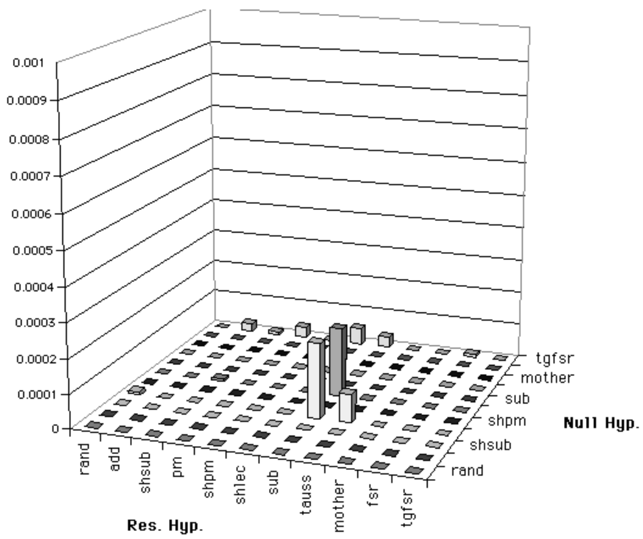

Figure 1: H_0 Rejections for Artificial Ant Problem

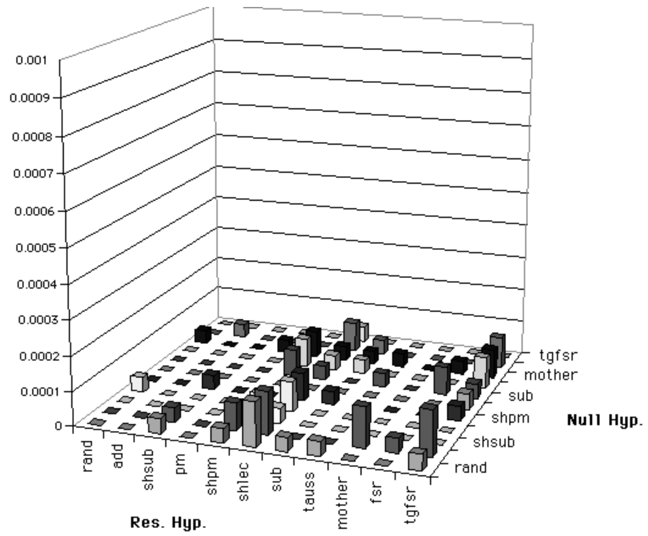

Figure 2: H_0 Rejections for 8×8 Lawnmower Problem

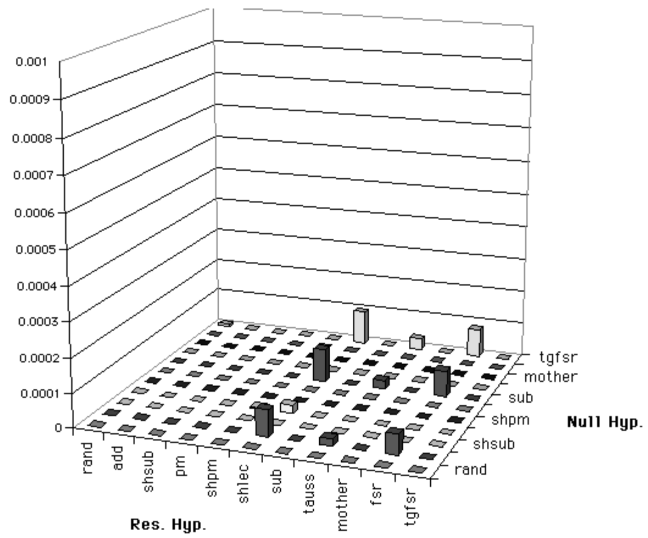

Figure 3: H_0 Rejections for Multiplexor Problem

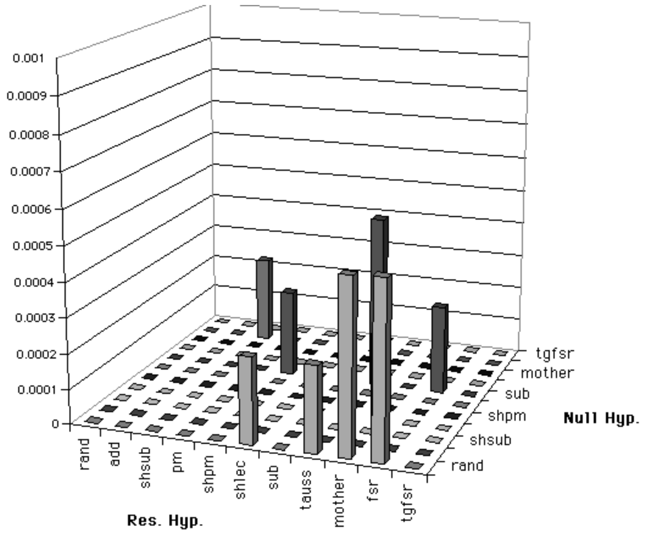

Figure 4: H_0 Rejections for Symbolic Regression Problem

5 CONCLUSIONS

We found very few null hypothesis rejections in our statistical examination of GP performance. The legends on Figures 1, 2, 3, and 4 show that the maximum normalized values in the figures were on the order of 10^{-4}, far from 1.0. So, we found no statistically significant evidence that improved PRNG quality caused improved GP performance. Further, there were no pronounced, complete rows or columns of null hypothesis rejections. Instead, the null hypothesis rejections we did find were scattered.

Although these results were from only a limited number of GP test problems, the results of this study were consistent with the results of our previous work. In this study, as in our GA studies, we found no statistical evidence that improved PRNG quality caused improved GP performance.

6 FURTHER RESEARCH

There are several other aspects of GP performance we would like to examine. For example, how does PRNG quality effect the complexity of individuals in the population, or the diversity of the population? We have data on these aspects; all that remains is to apply statistical analyses to the data. We would also like to expand our work to other GP test problems, and to add some theoretical weight to our studies, which have been purely statistical to this point.

References

[Daida et al., 1997] Daida, J., Ross, S., McClain, J., Ampy, D., and Holczer, M. (1997). Challenges with verification, repeatability, and meaningful comparisons in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 64–69, Stanford University, CA, USA. Morgan Kaufmann.

[Daida et al., 1999] Daida, J. M., Ampy, D. S., Raatanasavetavadhana, M., Li, H., and Chaudhri, O. A. (1999). Challenges with verification, repeatability, and meaningful comparison in genetic programming: Gibson's conundrum. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, FL, USA. Morgan Kaufmann.

[Green and Margerison, 1978] Green, J. R. and Margerison, D. (1978). *Statistical Treatment of Experimental Data*. Elsevier Scientific Publishing Company, first edition.

[Johnson, 1996] Johnson, B. C. (1996). Radix-b extensions to some common empirical tests for pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 6(4):261 – 273.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming*, volume 2. Addison Wesley, third edition.

[Kokkonen, 1997] Kokkonen, K. (1997). http://www.pcisys.net/ kimk/gpjpp.htm.

[Koza, 1992a] Koza, J. R. (1992a). Evolution of subsumption using genetic programming. In Varela, F. and Bourgine, P., editors, *Proceedings of the First European Conference of Artificial Life: Towards a Practice of Autonomous Systems*, pages 110 – 119. The MIT Press.

[Koza, 1992b] Koza, J. R. (1992b). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press.

[Koza, 1994] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press.

[Marsaglia, 1993] Marsaglia, G. (1993). Monkey tests for random number generators. *Computers & Mathematics with Applications*, 9:1–10.

[Marsaglia, 1994] Marsaglia, G. (1994). Yet another rng. Posted to sci.stat.math August 1, 1994.

[Marsaglia, 1998] Marsaglia, G. (1998). http://stat.fsu.edu/~geo/diehard.html.

[Matsumoto and Kurita, 1992] Matsumoto, M. and Kurita, Y. (1992). Twisted gfsr generators. *ACM Transactions on Modeling and Computer Simulation*, 2(3):179 – 194.

[Meysenburg, 1997] Meysenburg, M. M. (1997). The effect of pseudo-random number generator quality on the performance of a simple genetic algorithm. Master's thesis, University of Idaho.

[Meysenburg and Foster, 1997] Meysenburg, M. M. and Foster, J. A. (1997). The quality of pseudo-random number generators and simple genetic algorithm performance. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 276 – 281. Morgan Kaufmann.

[Press et al., 1992] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press, second edition.

[Schneier, 1994] Schneier, B. (1994). *Applied Cryptography*. John Wiley And Sons.

[Tezuka and L'Ecuyer, 1991] Tezuka, S. and L'Ecuyer, P. (1991). Efficient and portable combined tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1(2):99 – 112.