A comparative computational study of random number generators

Celso C. Ribeiro
Universidade Federal Fluminense, Department of Computer Science
Rua Passo da Pátria 156, Niterói 24210-240, Brazil
email: celso@ic.uff.br

Reinaldo C. Souza
Pontifícia Universidade Católica do Rio de Janeiro, Department of Electrical Engineering
Rua Marquês de São Vicente 225, Rio de Janeiro 22453-900, Brazil
email: reinaldo@ele.puc-rio.br

Carlos Eduardo C. Vieira
Pontifícia Universidade Católica do Rio de Janeiro, Department of Computer Science
Rua Marquês de São Vicente 225, Rio de Janeiro 22453-900, Brazil
email: cadu@inf.puc-rio.br

Abstract

Randomization plays a very important role in algorithm design. Metaheuristics such as simulated annealing, GRASP, genetic algorithms, and VNS make systematic use of randomization at different levels. Therefore, the use of consistent random number generators is highly recommended. We considered three well known generators and we investigated some of their properties. The three generators were submitted to two different classes of statistical tests. We conclude by showing some good properties of the Mersenne Twister generator that do not seem to be met by the others.

Keywords: Random numbers, generators, statistical tests, randomized algorithms.
AMS subject classification: 65C10, 68W20, 90C59.

1 Introduction

Random number generators simulate the abstract mathematical idea of independent uniformly distributed random variables over the interval $[0, 1]$. Random variables following other distributions can be simulated by appropriate transformations from uniform distributions [17]. The generators are specific algorithms that behave deterministically once initialized with the same seed. The generated numbers are, in fact, pseudo-random.

Randomization plays a very important role in algorithm design. In the context of optimization by metaheuristics, randomization can be used e.g. to break ties so as that different solution paths can be visited from the same initial solution on multi-start methods or to unbiased sample fractions of large neighborhoods. One particularly important use of randomization appears in the context of greedy randomized algorithms. The latter are based on the same principles of pure greedy algorithms, but make use of randomization to build different solutions at different runs. Greedy randomized algorithms are used e.g. in the construction phase of GRASP heuristics [6, 7, 33, 34] or to create initial populations for population methods such as genetic algorithms [13, 14, 35, 36]. Randomization is also a major component of metaheuristics such as simulated annealing [1, 2, 12] and VNS [9, 10, 11, 32], in which a solution in the neighborhood of the current one is randomly generated at each iteration.

Therefore, random number generators have a strong influence on the effectiveness of metaheuristics for optimization problems. In this work, we evaluate and compare three random number generators widely used in the implementation of metaheuristics: the old UNIX rand() generator implemented and used in the ANSI C language [37], the widely used generator proposed by Lewis et al. [24] and implemented in a portable way by Schrage [38], and the more recent Mersenne Twister generator of Matsumoto and Nishimura [31]. The three generators are described in Section 2. Two packages of statistical tests are applied to the three generators and experimental results are presented in Section 3. Concluding remarks are drawn in the last section.

2 Some random number generators

2.1 Old UNIX rand generator

The old UNIX rand() generator [37] referred by UR in this work is a *mixed linear congruential generator* defined as

$$x_{n+1} = (1103515245 x_n + 12345) \bmod (2^{31}), \tag{1}$$

where x_n is the n-th number in the generated sequence. This recursion generates integer numbers in the interval $[0, 2^{31} - 1]$.

2.2 Schrage's generator

The generator proposed by Lewis et al. [24], implemented in a portable way by Schrage [38] and referred by LS in this work is a popular *multiplicative linear congruential generator* defined as

$$x_{n+1} = 7^5 x_n \bmod (2^{31} - 1), \tag{2}$$

where x_n is the n-th number in the generated sequence and $2^{31} - 1 = 2147483647$ is Mersenne's prime number. This recursion generates integer numbers in the interval $[1, 2^{31} - 2]$. This is a *full cycle generator*, since $a = 7^5$ is a primitive root of $2^{31} - 1$. Therefore, every integer in the interval $[1, 2^{31} - 2]$ is generated exactly once in the cycle. The initial seed may be any integer in the interval $[1, 2^{31} - 2]$.

2.3 Mersenne Twister generator

The third algorithm considered in this work is the Mersenne Twister (MT) generator developed by Matsumoto and Nishimura [31]. It has a very large period equal to $2^{19937} - 1$, being a variant of the generator previously proposed in [29, 30]. We used the mt19937ar implementation available from the Mersenne Twister web site in [28].

3 Testing random number generators

Two classes of tests are often applied to evaluate random number generators [20]. Theoretical or structural tests take into account the structure of the generator to evaluate its mathematical properties, such as the period and lattice structure. Generators without good properties should be avoided [19].

We consider as an example the period of a generator. The period of a generator cannot exceed the cardinality of its state space. Consequently, the period should be as close as possible to the latter. Many generators satisfy this property if their parameters are appropriately chosen [16, 18], including the three considered in this work. The period of a generator imposes a limit on the size of the samples. L'Ecuyer [17, 20, 23] emphasizes that generators with a period close to 2^{31} should be discarded, since it can be exhausted in a few minutes of computation. He also emphasizes that periods lower than 2^{50} are small and that periods greater than 2^{200} should be used. Ripley [37] suggests periods of 2^{31} for samples with up to 10^3 points and of at least 2^{50} for samples with more than 10^6 points.

Statistical tests are used as a complement to the tests that consider structural properties. They evaluate the quality of long sequences produced by the generator [22]. Since the three generators considered in this work have good structural properties, this work is focused into their statistical evaluation.

The statistical tests consider the random number generators as black boxes and check their behavior against the null hypothesis H_0: the observations are independent and uniformly distributed in the interval $[0, 1]$. Their main idea consists in attempting to find situations in which the behavior of some output function of the generator is significantly different from that of the same function applied to a sequence of independent random variables uniformly distributed in the interval [0,1] [21].

We first applied classical tests extracted from [15, 16]. However, as several suspected generators have passed them [3, 22], we also applied the stronger Diehard tests developed by Marsaglia [26]. All tests were performed on a Pentium III machine with a 750 MHz clock and 256 Mbytes of RAM memory running under Linux RedHat 7.0. For each test and each generator, 100 runs using 100 different seeds have been performed.

3.1 Classical tests

We use the same notation and tests in Knuth [16]. Tests are applied to sequences $u_0, u_1, u_2, \ldots$ of real numbers or to sequences $y_0, y_1, y_2, \ldots$ of integer numbers, with $y_j = \lfloor d \cdot u_j \rfloor$ for every index j in the sequence. In the first case, we assume that the sequence is independent and uniformly distributed in the interval [0,1], while in the second it is assumed to be independent and uniformly distributed in the interval [0,$d - 1$]. The integer d is appropriately chosen. For instance, $d = 64 = 2^6$ implies that y_j represents the six most significant bits of u_j. The significance level considered in all tests is $\alpha = 0.05$.

1. Chi-square test in the interval [0,1]: A sequence of n random numbers $u_1, u_2, \ldots, u_n$ is generated in the interval [0,1], which is divided into k subintervals. We used $k = 5$, 10 and $n = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

2. Frequency test or chi-square test in the interval [0,d]: A sequence of n random numbers $y_1, y_2, \ldots, y_n$ is generated in the interval $[0, d)$. For every integer $r \in [0, d)$, we count the number of times $y_j = r$ for $j = 1, \ldots, n$ and we apply the chi-square test for d categories. We used $d = 32$, 64 and $n = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

3. Kolmogorov-Smirnov (KS) test: The random numbers $u_1, u_2, \ldots, u_n$ are sorted in non-decreasing order and the statistics $K^+ = \sqrt{n}\, max_j\ [j/n - x_j]$ and $K^- = \sqrt{n}\ max_j\ [x_j - (j-1)/n]$ are computed. We used $n = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000; see also [15].

4. Two-level test: This test consists in applying the Kolmogorov-Smirnov test to p samples of size N and then applying KS to the p statistics so obtained. We used $N = 1000$ and $p = 5$, 10, 50, 100, 500, 1000, 5000; see also [15].

5. Serial test: A sequence of $n = 2p$ random numbers $u_1, u_2, \ldots, u_n$ is generated in the interval [0,1]. For every pair of integers (q, r) with $0 \leq q, r < d$, we count the number of occurrences of the pair $(y_{2j+1}, y_{2j+2}) = (q, r)$, for $j = 0, \ldots, p-1$. We used $d = 2$, 3, 4, 5 and $p = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

6. Permutation test: The input sequence $u_1, u_2, \ldots, u_n$ is divided into g groups of t elements each, i.e. $n = gt$. Elements in each group may follow up to $t!$ possible relative orders. The number of times in which each relative order appears is counted. We used $t = 2$, 3, 4, 5 and $g =$ 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

7. Gap test: Let α and β be two real numbers with $0 \leq \alpha < \beta \leq 1$. This test considers the size of subsequences of consecutive numbers $u_j, u_{j+1}, \ldots, u_{j+r}$ in which the last is the only one in the interval $[\alpha, \beta]$. This subsequence of size $r+1$ is a gap of size r. Applied to a sequence of n real numbers in the interval [0,1] and for any values of α and β, this test counts the number of gaps of size $0, 1, \ldots, t-1$ and the number of gaps of size greater than or equal to t, until s subsequences have been found. We used $\alpha = 0.25$, $\beta = 0.5$, $t = 6$, and $s = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

8. Coupon collector's test: A sequence of n random numbers $y_1, y_2, \ldots, y_n$ is generated in the interval $[0, d)$. We count the number of minimal segments $y_{j+1}, y_{j+2}, \ldots, y_{j+r}$ of length $r = d, d+1, \ldots, t-1$ and the number of segments of length greater than or equal to t containing all integers in the interval $[0, d)$, until s segments have been found. We used $d = 3$, $t = 6$, and $s = 500$, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

9. Poker test: This test considers g groups of five successive integer random numbers $\{y_{5j+1}, y_{5j+2}, \ldots, y_{5j+5}\}$, for $j = 0, \ldots, g-1$, and counts the

number of groups with r different values. We used $d = 5$, $r = 5$ and $g =$ 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

10. Maximum of t test: Let $V_j = max(u_{tj}, u_{tj+1}, \ldots, u_{tj+t-1})$ for $0 \leq j < g$. This test consists in applying the KS test to the sequence $V_0, V_1, \ldots, V_{g-1}$ with the distribution function $F(x) = x^t$, for $0 \leq x \leq 1$. We used $t = 5$, 6, and $g =$ 5000, 10000, 50000, 100000, 500000, 1000000, 5000000.

11. Serial correlation test: The covariance between pairs u_j and u_{j+k} of numbers k positions away in the sequence should be a normal distribution with expected value equal to 0 and variance $1/[144(n-k)]$. We used $k = 1, 2, \ldots, 15$ and $n =$ 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000; see also [15].

In order to validate the random number generators on the classical tests, we used the well known statistical tool known as two-way analysis of variance (ANOVA) for sampling without replacement [39]. The two factors considered in the analysis are the generator and the sample size used in each test. Therefore, two hypotheses could be tested:

1. There is no significant difference among the generators.

2. There is no significant difference among the sample sizes.

If the two hypotheses are confirmed, one can also state that there is no relationship between the two factors. On the other hand, if they are not confirmed, one can check what was the cause of the dependency (the generator or the sample size). The following step in the analysis consists in the implementation of another statistical test known as the LSD method (Least Significant Difference), which points out the statistically significant means of the factor that has caused the dependency.

Table 1 shows the way the data (generators and sample sizes) are organized to perform the two-way ANOVA without replacement. The rows represent the generators ($i = 1, 2, 3$), the columns represent the sample sizes, and x_{ij} denotes the number of errors that happened in 100 statistical tests (with 100 seeds) of the generator i for the sample size j.

	1	2	$\ldots$	m
1	x_{11}	x_{12}	$\ldots$	x_{1m}
2	x_{21}	x_{22}	$\ldots$	x_{2m}
3	x_{31}	x_{32}	$\ldots$	x_{3m}

Table 1: Two-way analysis of variance for sampling without replacement.

Table 2 displays the results of the ANOVA procedure for the classical tests, where an "A" means acceptance of the null hypothesis and an "R" means rejection of the null hypothesis. Out of 34 tests, in 24 of them the null hypothesis was accepted. For these tests, one can conclude that there is no relationship between the generator and the sample size. Individually, one can also conclude that there is no significant difference of performance among the three generators on the test, as well as on the sample size within each test. Out of the ten tests

Tests	Parameters	Results
Chi-square	$k = 5$	A
	$k = 10$	A
Frequency	$d = 32$	R
	$d = 64$	R
Serial	$d = 2$	A
	$d = 3$	R
	$d = 4$	A
	$d = 5$	A
Poker	—	A
Permutation	$t = 2$	A
	$t = 3$	A
	$t = 4$	A
	$t = 5$	R
Maximum of t	$t = 5$	A
	$t = 6$	A
Coupon collector	—	A
Serial correlation	$k = 1$	A
	$k = 2$	R
	$k = 3$	A
	$k = 4$	R
	$k = 5$	A
	$k = 6$	A
	$k = 7$	A
	$k = 8$	R
	$k = 9$	A
	$k = 10$	A
	$k = 11$	R
	$k = 12$	A
	$k = 13$	A
	$k = 14$	R
	$k = 15$	A
KS	—	A
Two-level	—	R
Gap	$t = 6$	A

Table 2: Results of the ANOVA procedure for the classical tests.

where the null hypothesis was rejected, six of them had the generator as the cause of the dependency, while four had the sample size causing the dependency.

It is recommended, particularly for the tests where a significant difference of generator was detected, to apply the LSD test mentioned above aiming the indication of which means are statistically significant. Table 3 displays the results of the LSD procedure applied to the six tests, where a "YES" in columns Dif_{URLS} , Dif_{URMT}, and Dif_{LSMT} indicates a significant difference respectively among the UR and LS generators, the UR and MT generators, and the LS and MT generators, while a "NO" means the opposite.

Figures 1 and 2 display the means of the errors, making it easier the understanding of Table 3. For the serial test with $d = 3$ and the serial correlation test with $k = 2$, the MT generator is the responsible for the means differences and there is no significant difference from the statistical viewpoint between the UR and LS generators. Therefore, these two generators have a better performance than the MT generator in these tests. For the serial correlation test with $k = 4$, the generators LS and MT outperformed UR. Considering the serial correlation test with $k = 8$, there is a statistical difference between the generator UR and MT, with the former outperforming the latter. The LS generator is outperformed by the others in the case of the correlation test with $k = 11$, while MT outperformed the two others in the case of the frequency test with $d = 32$.

Since there is not a consistent predominance of one generator over the others in these tests, one can conclude that there is no significant difference among the three generators considered on the classical tests and their corresponding parameters and sample sizes.

Tests	Dif_{URLS}	Dif_{URMT}	Dif_{LSMT}
Serial ($d = 3$)	NO	YES	YES
Serial correlation ($k = 2$)	NO	YES	YES
Serial correlation ($k = 4$)	YES	YES	NO
Serial correlation ($k = 8$)	NO	YES	NO
Serial correlation ($k = 11$)	YES	NO	YES
Frequency ($d = 32$)	NO	YES	YES

Table 3: LSD method applied to the tests for which the cause of the dependency is the generator.

3.2 Diehard tests

The Diehard tests designed by Marsaglia [3, 22, 25] are considered stronger than the classical tests. Their output is a series of $p-$values that should be uniform in $[0, 1)$. Those $p-$values are obtained by $p = F(X)$, where F is the assumed distribution of the sampled random variable X, which is often normal. When a bit stream really fails, the $p-$values will be of 0 or 1 to six or more decimal places. Moreover, these tests also apply the Kolmogorov-Smirnov test to the $p-$values themselves. If the null hypothesis of the KS test is rejected, the generator fails the test [26]. As for the classical tests, 100 seeds have been used.

The following tests were applied: birthday spacings test, greatest common divisor (GCD) test, gorilla test, overlapping 5-permutation test, binary rank test,

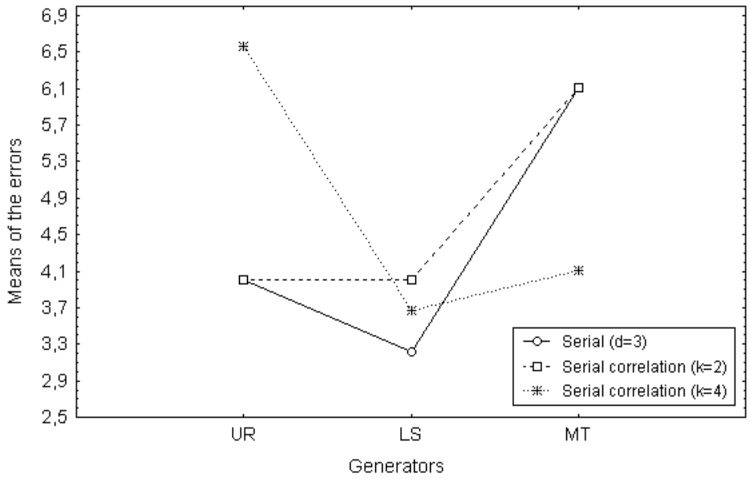

Figure 1: Means of the errors: serial test with $d = 3$ and serial correlation test with $k = 2$ and $k = 4$.

bitstream test, overlapping-pairs-sparse-occupancy (OPSO) test, overlapping-quadruples-sparse-occupancy (OQSO) test, DNA test, count-the-1's test, count-the-1's test for specific bytes, parking lot test, minimum distance test, 3D-spheres test, squeeze test, overlapping sums test, runs test, and craps test.

Table 4 displays the summary of the results obtained from the application of the Diehard tests to the UR, LS, and MT generators. A symbol "P" is an indication of success, i.e., the generator passed in the test, while an "F" indicates a failure in the test, i.e., the generator failed when submitted to the specific test.

The MT generator passed in all 23 tests of the battery, while the other generators passed in only five. It is interesting to notice that the choice of the starting seed does not pose any difference on the test, i.e., a failure (resp. success) of a generator with a seed in a particular test implies also in the failure (resp. success) of the generator for all other seeds of the same test. A particular standard also happens for the generators of the same class (UR and LS): they fail and pass in the same tests. For these generators, the results are very unsatisfactory on the lower order bits, showing that they are not random even when the module is a power of two (UR generator) or a prime number (LS generator). For example, for the OPSO, OQSO, and DNA tests, both generators failed on the bits from 1 to 10, 1 to 5, and 1 to 2, respectively. The MT generator showed an excellent statistical performance in all tests.

3.3 Speed and parallelization

Tables 5 and 6 display the processing times taken by each generator to produce samples of different sizes. For each sample size n, they give the average processing times for generators UR, LS, and MT over 100 runs with different seeds. Table 5 gives the average computation times in microseconds for smaller samples (for n ranging from 1000 to 1000000), while Table 6 gives the average computation times in seconds (s) for larger samples (for n ranging from 5000000

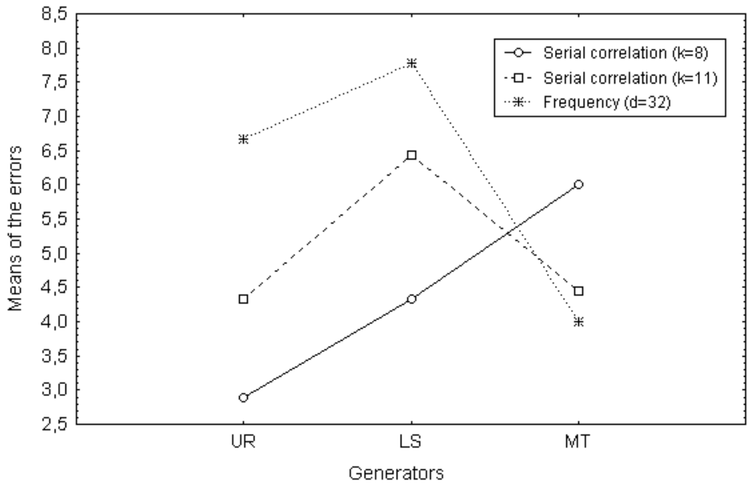

Figure 2: Means of the errors: correlation test (with $k = 8$ and $k = 11$) and frequency test.

to 2000000000). The same information is displayed in graphical form in Figures 3 and 4. We can see that the generator LS is considerably slower than the others, while MT seems to be slightly faster than UR.

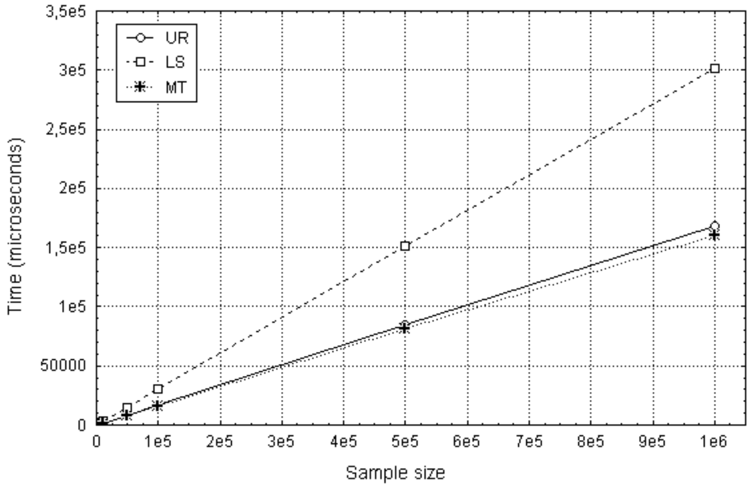

Figure 3: Processing times for the smaller samples.

Metaheuristics offer a wide range of possibilities for effective parallel algorithms running in much smaller computation times, but requiring efficient implementations. Cung et al. [4] and Martins et al. [27] showed that parallel implementations of metaheuristics appear quite naturally as an effective approach to speedup the search for good solutions to optimization problems. They lead to more robust algorithms, less-dependent on parameter tuning and not limited to few or small classes of problems. However, developing and tuning efficient

Tests	UR	LS	MT
Birthday spacings I	F	F	P
Birthday spacings II	F	F	P
GCD I	F	F	P
GCD II	P	P	P
Gorilla	F	F	P
Overlapping 5-permutation	P	P	P
Binary rank (31×31)	F	F	P
Binary rank (32×32)	F	F	P
Binary rank (6×8)	F	F	P
Bitstream	F	F	P
OPSO	F	F	P
OQSO	F	F	P
DNA	F	F	P
Count-the-1's	F	F	P
Count-the-1's for specific bytes	F	F	P
Parking lot	F	F	P
Minimum distance	P	P	P
3D-spheres	F	F	P
Squeeze	F	F	P
Overlapping sums	F	F	P
Runs	P	P	P
Craps I	F	F	P
Craps II	P	P	P

Table 4: Results of the Diehard tests.

parallel implementations of metaheuristics require a thorough programming effort.

In the case of parallel implementations of metaheuristics, the random number generators should ensure additionally that the sequences of random numbers generated at each processor should be independent one from the other [5]. Even reliable generators are not necessarily safe when used in parallel by multiple processors. There are basically two methods to generate random numbers on parallel processors [8]:

1. Assign p different generators to p different processors.

2. Assign p different subsequences of one large-period generator to p processors.

The first approach cannot be recommended, since in general there are no results on correlations between different generators. The second approach is not safe since some generators may produce bad or correlated subsequences. Generators with small periods are particularly dangerous to be used in parallel due to possible superpositions of subsequences.

Sample size n	UR	LS	MT
1000	207.74	316.34	226.56
5000	879.99	1520.61	888.54
10000	1722.91	3024.38	1690.08
50000	8489.52	15080.81	8300.72
100000	16876.70	30106.67	16199.46
500000	84737.59	150997.77	80861.47
1000000	168820.43	301479.76	160853.59

Table 5: Processing times for the smaller samples.

Sample size n	UR	LS	MT
5000000	0.86	1.53	0.79
10000000	1.68	3.00	1.63
20000000	3.33	6.02	3.21
50000000	8.36	15.02	8.05
100000000	16.65	30.02	16.08
200000000	33.35	60.04	32.18
500000000	83.33	150.05	80.37
1000000000	166.62	300.14	160.74
2000000000	333.25	600.18	321.51

Table 6: Processing times for the larger samples.

4 Concluding remarks

We evaluated the behavior of three random number generators. The analysis was mainly based on two sets of statistical tests: classical tests and the stronger tests in the Diehard package.

Large problem instances faced by state-of-the-art algorithms require the generation of very large samples of random numbers. The use of generators with large periods is strongly recommended. Generators with large periods are also recommended to be used in parallel algorithms, to avoid the repetition of subsequences in different processors.

The intrinsic nature of linear congruential generators leads to the appearance of lattice structures in $k-$dimensional spaces. They also show a cyclic behavior in low order bits, as confirmed by its rejection in the harder OQSO, OPSO, and DNA tests.

There are not significant differences between the three tested generators regarding the classical tests. However, the Mersenne Twister generator of Matsumoto and Nishimura [31] clearly outperformed the two others concerning the stronger Diehard tests. The algorithm used by the Mersenne Twister generator to generate the sequence of random numbers is also faster than those used by the other generators. Furthermore, its astronomical period favors large samples without repetitions, as well as its use in parallel implementations.

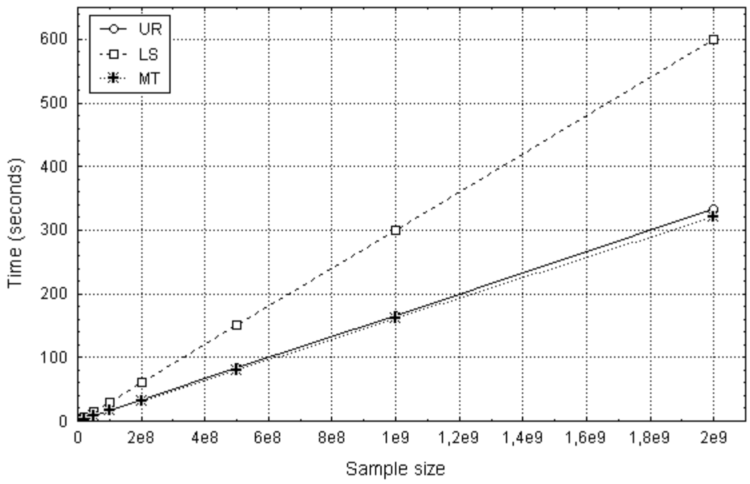

Figure 4: Processing times for the larger samples.

References

[1] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A stochastic Approach to Combinatorial Optimization and Neural Computing*, Wiley, 1989.

[2] E. Aarts and J. Korst, "Selected topics in simulated annealing", in *Essays and Surveys in Metaheuristics* (C.C. Ribeiro and P. Hansen, eds.), 1–37, Kluwer, 2002.

[3] S.L. Anderson, "Random numbers generators on vector supercomputers and other advanced architectures", *SIAM Rev.* 32 (1990), 221–251.

[4] V.-D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol, "Strategies for the parallel implementation of metaheuristics", in *Essays and Surveys in Metaheuristics* (C.C. Ribeiro and P. Hansen, eds.), 263–308, Kluwer, 2002.

[5] W.F. Eddy, "Random number generators for parallel processors", *J. Comput. Appl. Math.* 31 (1990), 63–71.

[6] T.A. Feo and M.G.C. Resende, "A probabilistic heuristic for a computationally difficult set covering problem", *Oper. Res. Lett.* 8 (1989), 67–71.

[7] T.A. Feo and M.G.C. Resende, "Greedy randomized adaptive search procedures", *J. Global Optim.* 6 (1995), 109–133.

[8] P. Hellekalek, "Good random number generators are (not so) easy to find", *Math. Comput. Simulation* 46 (1998), 485–505.

[9] P. Hansen and N. Mladenović, "An introduction to variable neighbourhood search", in *Metaheuristics: Advances and Trends in Local Search Procedures for Optimization* (S. Voss, S. Martello, I.H. Osman, and C. Roucairol, eds.), 433–458, Kluwer, 1999.

[10] P. Hansen and N. Mladenović, "Developments of variable neighborhood search", in *Essays and Surveys in Metaheuristics* (C.C. Ribeiro and P. Hansen, eds.), 415–439, Kluwer, 2002.

[11] P. Hansen and N. Mladenović, "Variable neighborhood search", in *Handbook of Metaheuristics* (F. Glover and G. Kochenberger, eds.), 145–184, Kluwer, 2003.

[12] D. Henderson, S.H. Jacobson, and A.W. Johnson, "The theory and practice of simulated annealing", in *Handbook of Metaheuristics* (F. Glover and G. Kochenberger, eds.), 287–319, Kluwer, 2003.

[13] J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[14] J.H. Holland, "Genetic algorithms", *Scientific American* 267 (1992), 44–50.

[15] R. Jain, *The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, 1991.

[16] D.E. Knuth, *The Art of Computer Programming, Volume 2 – Seminumerical Algorithms*, Addison Wesley, 2nd. edition, 1981.

[17] P. L´Ecuyer, "Random numbers", in *International Encyclopedia of Social and Behavioral Sciences* (N.J. Smelser and P.B. Baltes, eds.), 12735–12738, Pergamon Press, 2002.

[18] P. L´Ecuyer, "Random numbers for simulation", *Communications of the ACM* 33 (1990), 85–97.

[19] P. L´Ecuyer, "Uniform random number generation", *Ann. Oper. Res.* 53 (1994), 77–120.

[20] P. L´Ecuyer, "Testing random numbers generators", *Proceedings of the 1992IEEE Winter Simulation Conference*, 305-313, 1992.

[21] P. L´Ecuyer, "Random number generation", Chapter 4 in *Handbook of Simulation* (J. Banks, ed.), 93–137, Wiley, 1998.

[22] P. L´Ecuyer and P. Hellekalek, "Random number generators: Selection criteria and testing", *Lectures Notes in Statist.* 138 (1998), 223–266.

[23] P. L´Ecuyer, R. Simard, E.J. Chen, and W.D. Kelton, "An object-oriented random-number package with many long streams and substreams", *Oper. Res.* 50 (2002), 1073–1075.

[24] P.A. Lewis, A.S. Goodman, and J.M. Miller, "A pseudo-random number generator for the System /360", *IBM Systems Journal* 8 (1969), 136–146.

[25] G. Marsaglia, "A current view of random number generators", in *Computer Science and Statistics: The Interface* (L. Billard, ed.), 3–10, Elsevier, 1985.

[26] G. Marsaglia, "The Diehard battery of tests of randomness", online reference at http://stat.fsu.edu/pub/diehard, last visited on March 12, 2005.

[27] S.L. Martins, C.C. Ribeiro, and I. Rosseti, "Applications and parallel implementations of metaheuristics in network design and routing", *Lecture Notes in Comput. Sci.* 3285 (2004), 205–213.

[28] M. Matsumoto, "Mersenne Twister Home Page", online reference at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`, last visited on April 16, 2005.

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

[30] M. Matsumoto and Y. Kurita, "Twisted GFSR generators II", *ACM Transactions on Modeling and Computer Simulation* 4 (1994), 254–266.

[31] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", *ACM Transactions on Modeling and Computer Simulation* 8 (1998), 3–30.

[32] N. Mladenović and P. Hansen, "Variable neighbourhood search", *Computers and Operations Research* 24 (1997), 1097–1100.

[33] M.G.C. Resende and C.C. Ribeiro, "Greedy randomized adaptive search procedures", in *Handbook of Metaheuristics* (F. Glover and G. Kochenberger, eds.), 219–249, Kluwer, 2003.

[34] M.G.C. Resende and C.C. Ribeiro, "GRASP with path-relinking: Recent advances and applications", in *Metaheuristics: Progress as Real Problem Solvers* (T. Ibaraki, K. Nonobe, and M. Yagiura, eds.), 29–63, Kluwer, 2005.

[35] C.R. Reeves, "Genetic algorithms", in *Modern Heuristic Techniques for Combinatorial Problems* (C.R. Reeves, ed.), 151–196, Wiley, 1993.

[36] C.R. Reeves, "Genetic algorithms", in *Handbook of Metaheuristics* (F. Glover and G. Kochenberger, eds.), 65–82, Kluwer, 2003.

[37] B.D. Ripley, "Thoughts on pseudorandom number generators", *J. Comput. Appl. Math.* 31 (1990), 153–163.

[38] L. Schrage, "A more portable Fortran random number generator", *ACM Transactions on Mathematical Software* 5 (1979), 132–138.

[39] K.S. Trivedi, *Probability and statistics with reliability, queuing and computer science applications*, Prentice-Hall, 1992.