DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

ONDERZOEKSRAPPORT NR 9723

SHOULD WE USE A PORTABLE GENERATOR IN AN EMERGENCY?

by

Zeger DEGRAEVE
Linus SCHRAGE

Katholieke Universiteit Leuven

Naamsestraat 69, B-3000 Leuven

ONDERZOEKSRAPPORT NR 9723

SHOULD WE USE A PORTABLE GENERATOR IN AN EMERGENCY?

by

Zeger DEGRAEVE
Linus SCHRAGE

D/1997/2376/25

KATHOLIEKE UNIVERSITEIT LEUVEN

Department of Applied Economic Sciences

Should We Use a Portable Generator in an Emergency ?

by

Zeger Degraeve

Katholieke Universiteit Leuven, Department of Applied Economic Sciences, Belgium

zeger.degraeve@econ.kuleuven.ac.be

Linus Schrage

University of Chicago, Graduate School of Business, Chicago, IL, 60637, USA

linus.schrage@gsb.uchicago.edu

19 August 1997

Abstract

Problem generators are convenient tools for making large numbers of problem instances available to objectively evaluate the performance of different algorithms. We suggest that a) problem generators should be used only as a last resort, and b) if used they should be "portable", i.e., will generate the same problem instances on different computers, and c) use statistical methodology consistent with good experimental design. We provide a number of rules and tools to use when deciding to use a random problem generator.

Subject classifications : Tools for computational testing ; Problem generator ; Cutting Stock, Knapsack.

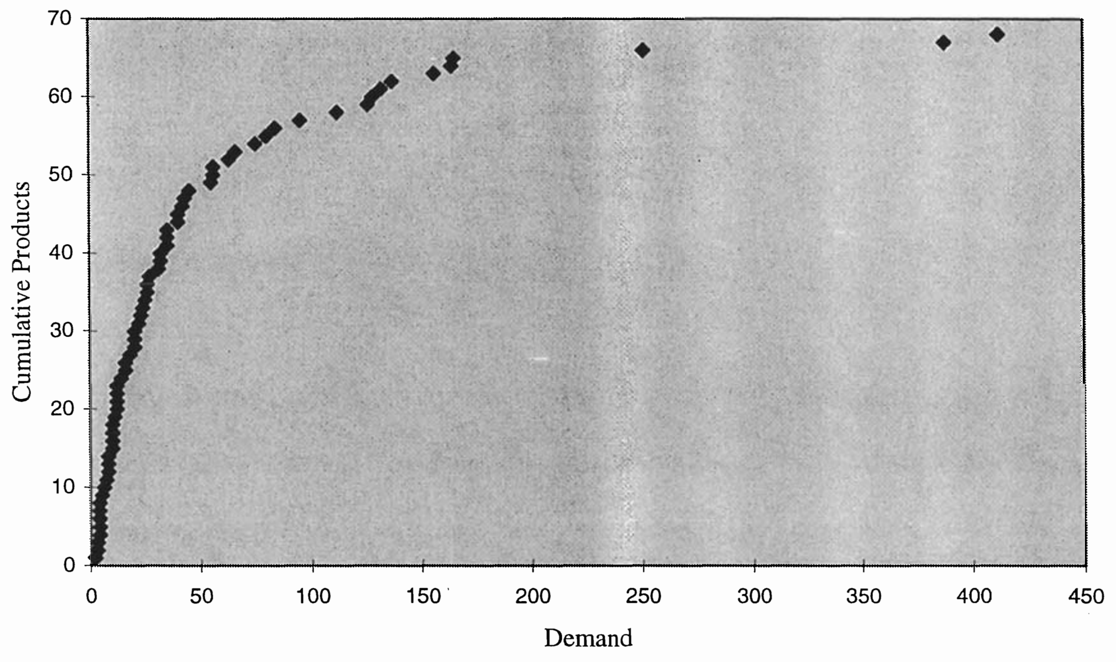
In order to objectively evaluate the performance of different algorithms, standard problem sets are frequently created. Distributing large numbers of large test problems is cumbersome. As an alternative, random problem generators have also been developed. A single, small problem generator may be able to generate a huge number of test problems. See the references for many examples of where random problem generators have been used to test algorithms for a wide range of problems. We make three arguments : a) random problem generators should be used only as a last resort, and b) if used, they should be portable in the sense that they give the same results on all computers, and c) they should be designed to be consistent with good experimental design.

1 Why Not Use a Random Problem Generator ?

We can think of three reasons for not using a random problem generator : a) the problems generated tend to not be representative of real industrial problems, b) solving lots of randomly generated problems gives a false sense of having thoroughly tested an algorithm, and, c) discouraging their use encourages us to go out and solve real problems.

To illustrate (a), consider the cutting stock problem as it occurs in the paper industry. One is given data on the number of rolls needed of each of a number of different widths of paper. The most common distributional assumption made in all random problem generators is that random variables have a uniform distribution. Gau and Wäscher (1995) describe a very comprehensive and flexible random problem generator for cutting stock problems. The generator used therein, assumes that demands for the various widths are uniformly distributed over some (input) interval. In Figure 1 is displayed the cumulative distribution for the amount demanded for each of 68 product widths for a paper cutting problem from a major paper manufacturer. The amounts demanded ranged from 2 to 412 units. Most of the demands fell between 2 and 200, but there were several between 200 and 412. It is clear that these demands are not uniformly distributed. It would be an improvement if a generator allowed a choice of non-uniform distributions to be used. Testing an algorithm on real empirical distributions would be even better.

Figure 1 : Cumulative Distribution of Product Demands.



To illustrate (b), consider the unbounded or general integer knapsack problem. See Babayev, Glover, and Ryan (1997), and Martello and Toth (1990) for excellent descriptions of the problem, as well as efficient algorithms for solving it. The form of the problem is :

$$\begin{aligned} \textit{Maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \textit{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le c \\ & x_j = 0, 1, 2, \ldots \end{aligned}$$

A plausible, and in fact common, way of generating random instances of this problem is to let p_j and w_j be random integers in [1, 1000]. Because the x_j are unbounded, variable j is dominated and can be set to 0 if there exists another k such that $w_j \ge w_k$ and $p_j \le p_k$. Because the w_j are integer, any problem with $n \ge 1000$ variables, can be reduced (in time proportional to n) to a problem with at most 1000 variables. Further, if c is not chosen randomly, (but is perhaps allowed to depend upon n), then, as n is allowed to increase, every randomly generated problem can be reduced in linear time, with probability approaching 1, to the single problem :

$$\begin{aligned} \textit{Maximize} \quad & 1000\, x_1 \\ \textit{Subject to} \quad & x_1 \le c \\ & x_1 = 0, 1, 2, \ldots \end{aligned}$$

Thus, even though we might ostensibly solve, say, 20 different big randomly generated problems, we are in fact only solving essentially one modest size underlying problem 20 times. This seems a less thorough test of an algorithm than one would like. Solving 20 different problems from 20 different industrial sources would be more reassuring.

We think reason (c) may be most important. If the operations research profession is to be successful, it is because it helps solve real problems, not imagined problems. One of the success stories of Operations Research is optimization. Linear and integer programming is widely used in industry. The substantial improvement in the performance of commercial optimization software on industrial problems in the last ten years coincides with the introduction and enhancement of the NETLIB and MIPLIB test data sets, see the websites : `http://www.mcs.anl.gov/home/otc/Guide/TestProblems/LPtest/index.html` and `http://softlib.rice.edu/softlib/catalog/miplib.html`. It is tempting to assume the latter caused the former. Almost every problem in these libraries is from some industrial setting. Almost every problem is unique in terms of its characteristics. Developing an algorithm that does well on every one of these diverse problems is a challenge.

2 How to Use Random Problem Generators if You Must

It is expensive and time consuming to collect real industrial problems. A random problem generator may be the only alternative if you want a problem with a particular characteristic, e.g. large size, quickly. In that case we think two general rules should be followed : a) the generator should be portable in the sense that some other researcher can run it on her machine and get essentially the same problem set, and b) the generator should be compatible with good statistical experimental design. We refine these into the following more specific rules : 1) input parameters to the problem generator should be integer, not real/floating point ; 2) use a good portable uniform random number generator to get the "raw" random numbers in the problem generator ; 3) transform the raw numbers with a monotonic transformation to get problem specific random numbers ; 4) transform the raw numbers in a portable fashion and 5) Make the entire problem generator portable.

We say a problem generator is portable if, given the same set of inputs, the generator produces the same set of outputs, to machine precision, when run on different computers.

Slightly more precisely, given a set of inputs $\{x_1, x_2,\ldots, x_m\}$, a problem generator on computer i produces the output $\{y_1^i, y_2^i,\ldots, y_n^i\}$. A generator is portable if there exists a small positive tolerance, ε, such that for any given input set $\{x_1, x_2,\ldots, x_m\}$, and for every two computers i and j, we have for every output term k that $|y_k^i - y_k^j| < \varepsilon * (1 + \max(\text{abs}(y_k^i), \text{abs}(y_k^j)))$. So in particular, if y_k^i and y_k^j are integers, then $y_k^i = y_k^j$. A major difficulty in writing portable software is that different kinds of computers do floating point arithmetic slightly differently. Almost all computer types do integer arithmetic identically. So the major technique in writing portable generators is to avoid floating point arithmetic in favor of integer arithmetic. We are now ready for rule 1.

Rule 1 : Input parameters for the problem generator should be in unambiguous integer format.

A common type of input to a problem generator is a fraction or probability, e.g., the fraction of the elements in a matrix that should be nonzero. It is natural to specify this number as a fraction, e.g., 0.4. Any computer, however, that uses binary, base 8, or base 16 floating point arithmetic (i.e., almost all popular computers), cannot represent 0.4 accurately. It is represented only approximately. Therefore, how it gets stored may vary among computers. So instead of allowing fractions directly as inputs, it is more portable to scale any such inputs, by, say 1000, and enter them as integers. Thus, 400 (out of 1000) is preferred to 0.4 as an input. For example, the cutting stock problem generator, CUTGEN1, from Gau and Wäscher (1995) requires the user to specify the range over which product widths are to be generated. This is done by specifying a raw material width, b, an integer, and two fractions, $0 < v_1 < v_2 \leq 1$, so that the product widths are in the interval $l_i \in [v_1 b, v_2 b]$. If the l_i are restricted to be integer, then the actual interval obtained may vary from computer to computer, depending upon how v_1b and v_2b are rounded. It is better to specify the two limits directly as integers.

Rule 2 : Use a good portable uniform pseudo-random number generator to get the "raw" random numbers.

Almost every mathematical software package contains a generator for uniform numbers in the interval (0,1). One should be reluctant to use these generators for two reasons, a) they are of questionable quality, and b) they are almost never documented, so one cannot generate the same stream of random numbers on some other system. As an alternative, one should use any of the good quality portable generators available in the public domain. CUTGEN1, for example, uses the portable uniform pseudo-random number generator described in Bratley, Fox, and Schrage (1987). An important advantage of these portable generators is that in addition to returning a floating point number in (0,1), they return a corresponding integer random number, typically uniformly distributed over the interval [1, $2^{31}-2$]. For portability, one typically wants to disregard the floating point number and use just the integer. The generator in Bratley, et. al. is a congruential generator using a multiplier of 16807. Although that generator is of good quality, Fishman and Moore (1986) recommend using the multiplier 742938285. A portable version using that multiplier is included in appendix A.

Rule 3 : Transform the raw uniforms with a monotonic transformation when generating the problem specific random variables.

To motivate this rule, consider the random number generation capabilities of the Excel spreadsheet program. The “Tools/Data analysis/Random number generation” menu item allows one to generate random numbers from a variety of distributions. One is able to specify an initial seed so that one can generate the same set of random numbers in different sheets. In Table 1, you can find sets of six random numbers generated from a uniform, normal, Poisson, and Bernoulli distributions, generated using Excel-95. In each of the four cases the initial seed was 55. Except for the Bernoulli case, the means were specified to be 15. The random numbers might be, say, demands placed on an inventory system.

Table 1 : Random Numbers Generated with Excel-95.

Uniform	Normal	Poisson	Bernoulli
10.0665	16.8761	14	1
12.7641	12.2333	15	1
16.3256	11.8622	13	0
14.6385	14.6887	12	1
15.8290	18.8710	8	0
19.5093	17.9562	12	0

In Table 2 are the results of doing exactly the same exercise in Excel-97.

Table 2 : Random Numbers Generated with Excel-97.

Uniform	Normal	Poisson	Bernoulli
10.0665	7.8212	14	1
12.7641	13.2787	15	1
16.3256	15.9820	13	0
14.6385	14.7369	12	1
15.8290	15.6071	8	0
19.5093	19.7963	12	0

Notice that the results are the same except for the Normal distribution. We argue that Excel-97 is an improvement over Excel-95, but not as much as it could be. When Excel-97 transforms "raw" uniforms into Normal or uniformly distributed random variables of arbitrary distribution, it apparently uses a monotonic increasing transformation. Notice that when a uniform random number from Excel-97 is large, then so is its corresponding Normal random variable. If we wished to test whether the choice of distribution makes a difference in problem difficulty, the data from Excel-97 would give a more unequivocal answer. The transformation to generate Poisson random variables does not appear to be a monotonic transformation. The method for generating Bernoulli's appears to be (unfortunately) a monotonic decreasing transformation. High values for the uniform and Normal outcomes implies that the corresponding Bernoulli random variable will be small, i.e., 0. We look forward to the release of Excel-99.

There is a second reason for recommending the use of a monotonic transformation. When a problem generator needs random values uniformly distributed over some specified interval, an alternative, the modulus (mod) transformation, is sometimes used to transform the raw uniform values into approximately uniformly distributed values into some other desired interval. Suppose the raws are uniform integers on $[1,17]$ and we want to transform them to approximately uniform on $[1,7]$. The results of the mod transformation :

$$output = 1 + \left(\left(input - 1 \right) \bmod 7 \right)$$

are given in Table 3. While the monotonic mapping is :

$$output = 1 + \left\lfloor 7 * \frac{input}{18} \right\rfloor,$$

resulting in a transformation illustrated in Table 4.

Table 3 : Result of the Mod Transformation.

Output	1	2	3	4	5	6	7
Mapping of	1	2	3	4	5	6	7
Input	8	9	10	11	12	13	14
	15	16	17				

Table 4 : Result of the Monotonic Transformation.

Output	1	2	3	4	5	6	7
Mapping of	1	3	6	8	11	13	16
Input	2	4	7	9	12	14	17
		5		10		15	

Comparing Tables 3 and 4, we observe that the monotonic transformation clearly looks more uniform that the mod transformation. The modulus transformation tends to over-represent small values, and under-represent large outcomes.

Rule 4 : Transform the raw uniforms generated in a portable fashion.

Floating point arithmetic should be avoided in favor of integer arithmetic when generating a discrete outcome, such as an integer, or the choice of a branch in some decision process. To satisfy rules 3 and 4 for the commonly used case of uniformly distributed integers, we provide the portable subroutine VUNIFI in appendix B.. Its essential task is to compute portably, an integer x, the integer part of $u * v / w$, i.e. :

$$x=\left\lfloor u*{}^{v}\!/\!_{w}\right\rfloor \tag{2.1}$$

The interpretations are : the raw random variables are integers uniform on [0, 1, ...w-1], v is a draw from that distribution, and the output is to be approximately uniform on [0,1,...,u-1]. For the method to work on typical 32 bit computers, we need :

$$4 < w \leq 2^{31}-1 \tag{2.2}$$

To avoid overflow, VUNIFI requires :

$$0 \leq u, \ (u+2) *(u+2) < w, \tag{2.3}$$

$$0 \leq v \leq w, \tag{2.4}$$

Restriction (2.3) does not seem a serious one. It essentially restricts one to random integers in the range [0, 46300].

The "obvious" way to compute (2.1) is to first do the multiplication in double precision, then do the division, and then take the integer part. The portability problems with the various steps of that approach are : a) the largest integer product can be computed exactly in double precision varies from computer to computer, b) how accurately the fractional part gets represented after doing the division varies from computer to computer, and therefore c) the integer part may get computed differently among different computers. To our knowledge, all popular computers that support 4-byte integers, produce identical results in integer arithmetic as long as there is no overflow. Therefore, VUNIFI uses all integer arithmetic, while avoiding overflow. The essential "trick" of VUNIFI is to first compute an upper bound on x :

$$b = \lfloor v / \lfloor w/u \rfloor \rfloor \tag{2.5}$$

Clearly, b is a valid upper bound because $\lfloor w/u \rfloor \le w/u$. Observe that proper use of brackets ensures that the results of all calculations can be represented exactly as 32 bit integers. Given the size restriction on u, it follows that b overestimates x by at most 2. VUNIFI checks for this overestimation and decreases b as needed. Details are given in the comments in the code.

In another recently published example, the random problem generator simply generated a sequence of uniformly generated real numbers in (0,1), where the major purpose of the input was to simply specify the number of such numbers to be generated. In this case, simply using the output from a portable generator, such as URAND, would have satisfied our definition of portability. What was done, however, to make the results "more random", was to instead use six different random number generators. To get the next random output random number, a random number was first drawn from a seventh random number generator. This number was then used to choose which of the six generators would be used to generate the next output number. Unfortunately, floating point arithmetic was used in making the random choice among the six. Different computer types might at some point in the sequence choose a different stream, simply because of round-off differences. Once this happened, subsequent numbers could be dramatically different from computer to computer. Two observations are pertinent : a) it is not clear that randomly choosing among random generators produces more truly random results, and b) if, nevertheless, one wishes to choose randomly among generators, a portable way of doing it is to use VUNIFI to generate a random integer uniform in [0, 5], and then use this as an index into the random generator to use for the next output draw.

Rule 5: Write standard compliant code in a portable standard high level programming language.

Even if the two portable subroutines mentioned above are used, things can still go wrong. Recently, when we tried to compile a published generator that was described as portable, the first attempt at compiling it failed because the program name was the same as the name of the principal subroutine, a violation of the standard for the language FORTRAN. Thus, one should not only choose a popular, standardized language, e.g., C or FORTRAN, but also adhere to the language standard when writing the generator. Another problem we encountered with a recent published generator resulted from inconsistency in the case used.

In one part of the program a variable was referred to as “z”, whereas in another part of the program, “Z” was used to refer to the same variable. Standard FORTRAN is case insensitive and considers these two symbols as referring to the same variable. Absoft FORTRAN on the Mac, however, deviates from the standard and considers these as two different variables. The moral is, be consistent and do not push the standard. A minor portability problem with the C language was that older, freely distributed compilers only recognized so-called Kernigan&Ritchie C, whereas, recent compilers are designed for the ANSI-standard version of C. A number of reliable translators, e.g., `f2c`, are available for converting standard FORTRAN programs into standard C programs.

3 Computational Examples

As an exercise, we have written a portable version, CUTGENP, of the CUTGEN1 generator, using the rules and tools mentioned above. We have compared CUTGENP with CUTGEN1 using a number of different compilers and computers as indicated in Table 5.

Table 5 : FORTRAN Compilers and Computers Used for the Experiments.

ID	Compiler	Computer and Operating System
Watcom	Watcom Fortran 77 Version 10.6	Dell Dimension XPS Pro 200n Windows 95
MSFort	Microsoft Fortran 77 Powerstation	Dell Dimension XPS Pro 200n Windows 95
XLF	IBM Fortran 77	IBM SP2 workstation RISC 6000 UNIX (AIX)
FortVS	IBM VS Fortran 77 Version 2 Release 6	IBM 9672/R51 mainframe CMS (VM)
HP	HP Fortran compiler	HP-700 Unix
Alpha	DEC Alpha Fortran	DEC Alpha Unix (OSF)
VMS	VMS Fortran	DEC Alpha VMS
Sparc	Sparc Fortran	Sparc-20 SparcOS
Linux	Linux/GNU Fortran	Gateway Pentium Pro Linux
SGI	Silicon Graphics Fortran	Silicon Graphics Unix (IRIX Sys V.4)
Mac	Absoft	Apple Macintosh Power 8100 System 7

CUTGEN1 uses floating point arithmetic at some crucial points in its procedure, even though it produces integer output. Thus, it may generate different output sets on different computers for the same set of input parameters. As expected, CUTGENP generates the same output on each computer for a given input set. We found that, for a given input data set, CUTGEN1 gave one set of results on Watcom, Linux and XLF, and another set of results (almost identical to CUTGENP) on MSFort, SGI, HP, VMS, Sparc, Mac, Alpha and FortVS. For reference, one set of input parameters that illustrates the non-portability is :

```
10   1000   .375   .625   50   991759555   150
```

with the second last number being the random number seed. For input to CUTGENP, the decimal points are dropped on the 3rd and 4th numbers, i.e., they are interpreted as fractions of 1000, the second input number.

4 Conclusions and Ideas for Future Work

A problem generator is only a valid alternative to making large numbers of test problems available to researchers to test new algorithms if it produces the same problem instances on different machines with different compilers. In this paper, we have identified five rules that should be considered in the development of portable generators of test problems. Future work can focus on checking how well other available problem generators in the literature satisfy our five rules and improving them if necessary such that they become really portable.

Appendix A : Random Number Generation Function URAND

```
      FUNCTION URAND( IX)
      INTEGER*4 IX
C  Portable random number generator using the recursion:
C     IX = 742938285 * IX MOD (2**(31) - 1)
C  using only 32 bits, including sign.
C
C  INPUTS:
C    IX = integer in the interval [ 1, 2147483646]
C
C  OUTPUTS:
C    IX = new (pseudorandom) integer in interval [ 1, 2147483646]
C    URAND = a uniform fraction in (0.0, 1.0).
C
C  COMMENTS:
C    Cycle length is 2147483646.  all integers in [1, 2147483646]
C    are generated exactly once in a cycle.
C
      INTEGER*4 K1
C  Date 7 Aug 1994 by L. Schrage
C
C  First note: 742938285 = 21309 * 34865
C   Compute IX = 21309 * IX MOD 2147483647
C    NOTE: 2147483647 = 21309 * 100778 + 5245
      K1 = IX / 100778
      IX = 21309 * ( IX - K1 * 100778) - K1 * 5245
      IF ( IX .LT. 0) IX = IX + 2147483647
C   Compute IX = 34865 * IX MOD 2147483647
C    Note: 2147483647 = 34865 * 61594 + 8837
      K1 = IX / 61594
      IX = 34865 * ( IX - K1 *  61594) - K1 * 8837
      IF ( IX .LT. 0) IX = IX + 2147483647
      URAND = IX * 4.656612875E-10
      RETURN
      END
```

Appendix B : SUBROUTINE VUNIFI

```
      SUBROUTINE VUNIFI( M, IY, IR, IX)
      INTEGER*4 M, IY, IR, IX
C
C  Portable routine to compute
C  integer part OF IY * IR / M
C
C  INPUTS :
C    M  = THE 'MODULUS', 4 < M < 2 ** 31
C    IR = THE 'RANGE',  0 < (IR+2) * (IR+2) < M
C    IY = A 'RANDOM INTEGER',  0 <= IY < M
C
C  It is caller's responsibility to check
C  the bounds on the inputs
C
C  OUTPUTS :
C    IX = GREATEST INTEGER <= IY * IR / M
C
C  E.G., If IY is uniform on [ 1, M-1]
C        Then IX is approximately uniform on [0, IR - 1]
```

```
C
C   1 August 1997   by Z. Degraeve and L. Schrage
      INTEGER*4 K, ID, IDNOM, INUMR, ITEMP, IXL
C
C  Get an upper bound on [ IY * IR/ M] = [ IY/( M/ IR)]
      K = M / IR
      IX = IY / K
      ID = M - K * IR
C
C  Now reduce IX if necessary.  The initial IX
C   overestimates the true/final value by at most 2
C   if M > 4
      DO 2090 IXL = IX, 0, -1
C
C  Test if IXL is small enough, i.e. in infinite precision:
C  IXL <= IY * IR / M, or
C  IXL * M <= IY * IR, or
C  IXL * ( K * IR + ID) <= IY * IR, or
C  IR * ( IY - IXL * K) >= IXL * ID, or
C    IY - IXL * K > 0 and
C  IR  >= IXL * ID / ( IY - IXL * K)
C  Note IXL <= IR + 2, and ID < IR, so no overflow
      INUMR = IXL * ID
      IF( INUMR .EQ. 0) GO TO 9000
C  Note, IXL <= IY/ K, so IXL * K <= IY, so no overflow
      IDNOM = IY - IXL * K
      IF( IDNOM .LE. 0) GO TO 2090
      ITEMP = INUMR/ IDNOM
      IF( IR .GT. ITEMP) GO TO 9000
      IF( IR .LT. ITEMP) GO TO 2090
C  We have IR = ITEMP, so
C  IR  >= IXL * ID/ ( IY - IXL * K) only if no truncation
C  occurred in computing ITEMP.
C  Note, ITEMP <= INUMR/ IDNOM, so no overflow
        IF ( ITEMP * IDNOM .GE. INUMR) GO TO 9000
 2090 CONTINUE
C
 9000 IX = IXL
C
      RETURN
      END
C
```

References

Agrawal, M.K., Elmaghraby, S.E. and W.S. Herroelen, 1996, "DAGEN : A generator of Test Sets for Project Activity Nets", *European Journal of Operational Research*, 90, 376-382.

Arthur, J.L. and J.O. Frendewey, 1988, "Generating Travelling-Salesman Problems with Known Optimal Tours", *Journal of the Operational Research Society*, 39(2), 153-159.

Babayev, D., F. Glover, and J. Ryan, 1997, "A New Knapsack Solution Approach by Integer Equivalent Aggregation and Consistency Determination", *INFORMS Journal on Computing*, 9(1), 43-50.

Bratley, P., B. Fox, and L. Schrage, 1987, *A Guide to Simulation*, Springer-Verlag, New York.

Degraeve, Z. and L. Schrage, 1997, "Optimal Integer Solutions to Real-Life Cutting-Stock Problems", Research Report, Katholieke Universiteit Leuven, Department of Applied Economic Sciences, Belgium.

Degraeve, Z., 1992, "Scheduling Joint Product Operations with Proposal Generation Methods", Ph.D. Dissertation, The University of Chicago, Graduate School of Business, Chicago, Illinois.

Demeulemeester, E., Dodin, B. and W. Herroelen, 1993, "A Random Activity Network Generator", *Operations Research*, 41(5), 972-980.

Fishman, G. and L. Moore, 1986, "An Exhaustive Analysis of Multiplicative Congruential random Number Generators with Modulus 2^{31}-1", Siam Journal of Scientific and Statistical Computation", vol. 7, no. 1, pp. 24-45.

Gau, T. and G. Wäscher, 1995, "CUTGEN1 : A Problem Generator for the Standard One-Dimensional Cutting Stock Problem", *European Journal of Operational Research*, 84, 572-579.

Greenberg, H.J., 1991, "RANDMOD: A System for Randomizing Modifications to an Instance of a Linear Program", *ORSA Journal on Computing*, 3(2), Spring, 173-175.

Gilmore, P.C. and R.E. Gomory, 1961, "A Linear Programming Approach to the Cutting-Stock Problem", *Operations Research*, 9, 849-859.

Klingman, D., Napier, A. and J. Stutz, 1974, "NETGEN : A Program for Generating Large Scale Capacitated Assignment, Transportation, and Minimum cost Flow Network Problems", *Management Science*, 20(5), 814-821.

Kolisch, R., Sprecher, A. and A. Drexl, 1995, "Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems", *Management Science*, 41(10), 1693-1703.

Lin, B.W.Y., and R.L. Rardin, 1977, "Development of a Parametric Generating Procedure for Integer Programming Test Problems", *Journal of The Association for Computing Machinery*, 24(3), 465-472.

Martello, S. and P. Toth, 1990, "An Exact Algorithm for Large Unbounded Knapsack Problems", *Operations Research Letters*, vol. 9, pp. 15-20.

Sun, M., and R. Steuer, 1996, "Quad-Trees and Linear Lists for Identifying Nondominated Criterion Vectors", *INFORMS Journal on Computing*, vol. 8, no. 4, pp. 367-375.