On Generation of Test Problems for Linear Programming Codes

A. Charnes
University of Texas, Austin
W.M. Raike
The Naval Post Graduate School, Monterey
J.D. Stutz
University of Texas, Austin
and
A.S. Walters
Carnegie-Mellon University

Users of linear programming computer codes have realized the necessity of evaluating the capacity, effectiveness, and accuracy of the solutions provided by such codes. Large scale linear programming codes at most installations are assumed to be generating correct solutions without ever having been "bench-marked" by test problems with known solutions. The reason for this failure to adequately test the codes is that rarely are there large problems with known solutions readily available. This paper presents a theoretical justification and an illustrative implementation of a method for generating linear programming test problems with known solutions. The method permits the generation of test problems that are of arbitrary size and have a wide range of numerical characteristics.

Key Words and Phrases: linear programming, test problem generation, LP program evaluation, LP program validation

CR Categories: 5.41

Address of A.S. Walters: Office of the Dean, School of Urban and Public Affairs, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213.

1. Introduction

Since the earliest computational experiments with linear programming, users of linear programming computer codes have realized the necessity of evaluating the capacity, effectiveness, and accuracy of the solutions provided by such codes. Even today, this question of adequately "bench-marking" even the most thoroughly debugged codes arises, of course, in a variety of applications of computers to mathematics and scientific problems. The nature of meaningful linear programming problems, however, involves quite large matrices (say 2000 by 4000 with .6 to 2 percent density of nonzero elements) and consequently the data handling and generation problems become severe even if such problems with well-known computational characteristics can be found for test purposes. For this reason, tasks like the comparison of performance of linear programming codes must be carried out with the raw materials (i.e. test problems) at hand. Such comparisons are, of necessity, incomplete. It is our purpose here to present the (very elementary) theoretical justification and an illustrative, if rudimentary, implementation of a method for generating linear programming test problems with *known* solutions. The method permits the generation of test problems that are of arbitrary size and have an extremely wide range of numerical characteristics.

It is not our claim, of course, to have solved all the problems of testing linear programming codes. The developmental and the validation phases in the construction of large-scale linear programming codes present numerous challenges for adequately testing the various *parts* of the codes. While these areas, too, are virtually untracked territory, good programming practice and modern project management techniques can help to avert some fiascoes which are undocumented but which have become legendary. (An early example is the "nut mix code," so called because of its ability to speedily solve this textbook problem, which had been used as a test problem during development of the code, and to solve no other problems.) The area wherein our contribution lies is that of providing yardsticks for the gross evaluation of a number of overall performance characteristics of reasonably well-debugged codes. Perhaps one of the most useful applications of our technique could be expected to be that of measuring the solution time and the accuracy of some well-known and widely used linear programming systems when employed to solve very large problems.

Finally, the availability, in quantity, of a meaningful variety of test problems may help to influence the imple-

mentation of new solution techniques for the general linear programming problem. All too often, an elegant theory, rather than having gone hand-in-hand with effective performance in practice of new algorithms, has been a substitute for the performance.

2. Theoretical Basis

The mathematical observation underlying our procedures is not too deep. The main idea is that it is possible to construct a linear programming problem whose solution is obvious by inspection and then to restate the problem more generally without changing the set of feasible solutions. "Generally" here means that the solution would no longer be obvious were it not for the knowledge about the origin of the restated problem.

Consider the linear programming problem

$$\begin{aligned} &\max \sum_{j=1}^{n} d_j y_j \\ &\text{subject to: } \textstyle\sum_{j=1}^{n} a_j y_j = K \\ &0 \le y_j \le M_j, \quad \text{for } j = 1, \ldots, n. \end{aligned} \tag{1}$$

In (1), all d_j, a_j, and M_j as well as K are assumed to be positive. It is clear that no loss of generality is caused by assuming all $a_j = 1$, so henceforth we should consider the problem

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ &\text{subject to: } \textstyle\sum_{j=1}^{n} x_j = K \\ &0 \le x_j \le U_j, \quad \text{for } j = 1, \ldots, n. \end{aligned} \tag{2}$$

Or, restated in matrix notation,

$$\begin{aligned} &\max cx \\ &\text{subject to: } ex = K \\ &0 \le x \le U. \end{aligned} \tag{3}$$

In (2) and (3), the quantities c_j and U_j correspond to d_j/a_j and M_j/a_j from (1); the vector e is a vector of ones, and juxtaposition of vectors and/or matrices will be used throughout to denote the usual multiplications. Again with no loss of generality, we shall assume that $c_1 > c_2 > \cdots > c_n$.

An optimal solution, x^*, to (3) is easily obtainable by inspection; let $x_1^* = \min(K, U_1)$, define $K_1 = K - x_1^*$, and then calculate successive x_j^* and auxiliary K_j variable via the relations $x_j^* = \min(K_{j-1}, U_j)$, $K_j = K_{j-1} - x_j^*$. However, now adjoin slack variables, s, in (3). The resulting system is

$$\begin{aligned} &\max cx + 0s \\ &\text{subject to: } ex + 0s = K \\ &Ix + Is = U \\ &x, s \ge 0 \end{aligned} \tag{4}$$

If now we allow G to be an arbitrary r by n matrix (with $r \ge n$) which has full column rank, then the set of equations obtained by premultiplying the U constraints in (4) by G (namely $Gx + Gs = GU$) has exactly the same solution set as $Ix + Is = U$ since any such matrix G has at least one left inverse $G^{\#}$. The linear programming problem which results from using the new constraints is

$$\begin{aligned} &\max (c, 0) \begin{pmatrix} x \\ s \end{pmatrix} \\ &\text{subject to: } \begin{pmatrix} e & 0 \\ G & G \end{pmatrix} \begin{pmatrix} x \\ s \end{pmatrix} \begin{pmatrix} K \\ GU \end{pmatrix} \\ &x, s \ge 0. \end{aligned} \tag{5}$$

The solution x^* is not so obvious any longer (unless one happens to know the left inverse $G^{\#}$)! In fact, a slight additional complication could be introduced into (5) by allowing G to be r by $(n+1)$ and premultiplying all the equations in (4) by such a G.

The point of what we have done should now be apparent. Since the matrix G is completely arbitrary (except for the restriction to linearly independent columns), it can be chosen to embody any particular numerical characteristics desired. Thus, by controlling such aspects as the proportion of nonzero entries in G and the range (in orders of magnitude, say) of these nonzero entries, it is possible to construct a linear programming problem (5) having desired properties with regard to density, scaling, matrix size, etc. On the other hand, any special matrix structures (e.g. incidence matrices and block-angular matrices) can be imparted to (5) via G; the only inherent limitations are that G be of full column rank, that the constraint matrix have the form

$$\begin{pmatrix} e & 0 \\ G & G \end{pmatrix}$$

and that GU represent a positive linear combination of the columns of G.

3. Implementation

The code we present here is written in Fortran and was originally designed for use on the Control Data 6600 computer system of the University of Texas; with detail changes the program should be adaptable to most computers having magnetic tape capability and a Fortran compiler.

Objectives which the code was intended to accomplish include the following:

(i) The construction of a linear programming problem having a predetermined point as its optimal solution.

(ii) The ability to provide a constraint matrix of any desired size (within the theoretical limitation that G have full column rank). If an m by p final matrix is desired, then G must have $r = m - 1$, and if the column rank of G is to be full, $p \le 2(m-1)$.

(iii) The ability to control the density (proportion of nonzero entries) of the constraint matrix.

(iv) The ability to control the range of the magni-

tude of the nonzero matrix entries. We provide for input of a "scaling parameter" s which dictates that the matrix entries generated must fall between 10^{-s} and 10^{s} in absolute value.

(v) The ability to generate problems in a format suitable as input to the particular linear programming code being tested. For our purposes, this meant providing an alpha-numeric tape acceptable to CDC's OPTIMA system. We therefore decided to generate the constraint matrix one row at a time and to tailor the code's output section so that an OPTIMA input file could be created directly. For this purpose we are grateful to have had the use of the ACE II editing and matrix generator routine.[1] Since input sections of major linear programming codes tend to have their idiosyncrasies, it will be essential to tailor the format of the generated data to fit the particular code being tested. Having the matrix in standard constraint form does seem, however, to be acceptable to a number of commonly used matrix generator routines, and it is certainly readily recognizable by those wishing to use the algorithm.

We elected to provide a number of alternatives in the code. First, if no optimal solution is specified by the user, then the code will construct a problem having all $x_j = 1$, $S_j = 1$, and $U_j = 2$ (using the notation of (5)). The matrix G can be either supplied by the user or generated by the code. In the latter case, G is generated row-by-row in a pseudo random fashion in such a way as to embody all specified characteristics alluded to above. On the other hand, no special structure is provided for G in this case except for ensuring that the principal diagonal entries are nonzero. As a final alternative, an option is provided for generating an $m - 1$ by $m - 1$ Hilbert segment for G since such an ill-conditioned matrix provides numerous possibilities for difficulties in most matrix inversion routines.

4. Computer Listing with Comments to User

Sections of comment are interspersed for the purpose of explanation. These are not part of the code and are denoted by - - - before and after each section.

```
      PROGRAM GLUPP(INPUT,OUTPUT,TAPE1,TAPE2=OUTPUT,TAPE3,TAPE4,TAPE5,TA
     1PE6=TAPE1 ,TAPE7=INPUT)
      GLUPP GENERATES LP TEST PROBLEMS WITH KNOWN SOLUTIONS.
      COMMON/TOUGH/ M,N,RHO,S,AK,NNZ,NO,NV,ANR,ANS,PROP
      COMMON/BALLIT/X(5000),U(5000),GI(5000),C(5000)
      COMMON/GOOF/A,BP,CC,KDUZ
      CALL RDN
      CALL CRE8
      CALL ACE
      END
```

- - - If a matrix generator is being used, it should receive input from the algorithm on tape 6. Also, the call to ACE should be replaced by a call to the matrix generator being used (set up as a subroutine). All common throughout the algorithm is labeled common; thus unlabeled common in a matrix generator being linked to it would not have to be modified. If a matrix generator is not being used, either the call to ACE can be eliminated or a return jump can be executed in a dummy subroutine; and tape 6 should be set equal to output. This will print out the generated matrix in standard constraint form, and it can then be prepared for input to the linear programming code being used. - - -

```
      SUBROUTINE RDN
      COMMON/TOUGH/ M,N,RHO,S,AK,NNZ,NO,NV,ANR,ANS,PROP
      COMMON/BALLIT/X(5000),U(5000),GI(5000),C(5000)
      COMMON/GOOF/A,BB,CC,KDUZ
    1 FORMAT(2I5,2F5.1,I1,I2)
C     READ M=NUMBER OF ROWS, N=NUMBER OF COLUMNS IN LP PROBLEM (IF N IS
C     ODD IT WILL BE INCREASED BY 1),S= SCALING PARAMETER (S=2 IS O.K.),
C     RHO =APPROXIMATE DENSITY IN PERCENT,NO=1 IF SOLUTION WITH XJ=1 IS
C     WANTED OR NO =2 IF THE DESIRED OPTIMAL SOLUTION X IS TO BE READ
C     FROM CARDS.
C     KDUZ IS NEGATIVE IF
C     HILBERT SEGMENT IS WANTED, ZERO
C     IF MATRIX IS TO BE CREATED INTERNALLY,
C     POSITIVE IF MATRIX G IS SUPPLIED.
```

- - - The minimum required parameters supplied by the user would be the following case.

(i) In the first 5 columns using *I* format put the number of rows desired.

(ii) In columns 5–10, using *I* format, put the number of columns desired. (Remember n(columns 5–10) must be less than or equal to twice m (columns 1–5) $-$ 1, unless the Hilbert segment is wanted; in which case m must equal n.)

(iii) In columns 11–15, using *F* format, state density (in percent) desired. One decimal place is allowed (e.g. 10 .7 or 2.1 or .2).

(iv) In columns 16–20, using *F* format, put scaling factor desired (e.g. 2.0 will give nonzero entries between -100 and $-1/100$, and between $1/100$ and 100).

(v) In column 21 place a 1. This will give a solution vector of all ones.

(vi) In columns 22–23 place a -1 if the Hilbert segment is wanted (remember, this requires that m = n) or a zero, if not.

For the benefit of a user not familiar with it, a 3×3 Hilbert segment is

1/2 1/3 1/4
1/3 1/4 1/5
1/4 1/5 1/6. - - -

It should be pointed out that this program generates a problem assuming that the LP code to be used is, like OPTIMA, a *minimizing* code.

```
      SUBROUTINE GGEN(I)
      COMMON/TOUGH/ M, N,RHO,S,AK,NNZ,NO,NV,ANR,ANS,PROP
      COMMON/BALLIT/X(5000),U(5000),GI(5000),C(5000)
      COMMON/GOOF/A,BB,GC,KDUZ
      IF(KDUZ)3,1,2
    2 READ(7,6) (GI(J),J=1,NV)
      IF(EOF,7)9,8
    6 FORMAT(5E12.7)
    8 RETURN
    9 WRITE(2,11)
   11 FORMAT(52H INSUFFICIENT DATA FOR G MATRIX. GENERATION ABORTED.   )
      CALL EXIT
    1 XX=0.0
      DO 500 J=1,NV
      IF(J-1)100,10,100
   10 D=A*RANF(XX)-BB
      IF(ABS(D)-CC)10,20,20
   20 ITEMP=(D-IFIX(D))*10000+SIGN(.5,D)

      D=ITEMP/10000+IFIX(D)
      GI(J)=D
      GO TO 500
```

[1] ACE II was developed by William Briggs, who at the time was with Control Data Corporation.

```
100 ANS=ANS-2.0
    D=RANF(XX)
    IF(D -PROP)120,120,110
110 GI(J)=0.0
    IF(ANS)101,102,101
101 PROP=ANR/ANS
    GO TO 500
102 PROP=1.0
    GO TO 500
120 ANR=ANR-2.0
    IF(ANS)103,104,103
103 PROP=ANR/ANS
    GO TO 10
104 PROP=1.0
    GO TO 10
500 CONTINUE
    RETURN
  3 DO 5 J=1,NV
  5 GI(J)=1/(I+J)
    RETURN
    END
  2 FORMAT(6E12.7)
  3 FORMAT(1H1/1H ,19HPROBLEM PARAMETERS=/1H/,10X,3HM= ,20X,3HN= ,20X,
   19HDENSITY= )
  4 FORMAT(1H+,13X,I5,18X,I5,24X,E20.7)
    READ 1,M,N,S,RHO,NO,KDUZ
    NV=N/2
    IF(2*NV-N)5,10,10
  5 N=N+1
    NV=NV+1
 10 WRITE(2,3)
    WRITE(2,4)M,N,RHO
    IF(N-10000)101,101,999
101 IF(NV-M)9,9,999
999 WRITE(2,6)
    GO TO 9999
  6 FORMAT(72HPROBLEM GENERATION ABORTED; MUST HAVE N .LT. 10000 AND M
   1.GT. ONE-HALF N)
998 WRITE(2,7)
    GO TO 9999
  7 FORMAT(43HPROBLEM GENERATION ABORTED; DENSITY TOO LOW)
  9 IF(NO-1)200,200,100
100 READ 2,(X(J),J=1,NV)
    GO TO 210
200 DO 201 J=1,NV
201 X(J)=1.0
210 NNZ=.01*RHO*M*N-NV
    IF(KDUZ)211,212,212
211 M=NV+1
    WRITE(2,213)M
213 FORMAT(67H HILBERT SEGMENT GENERATION REQUIRES SQUARE G MATRIX, M
    CHANGED TO ,I5)
212 ANR=NNZ-N
    ANS=M*(M-1)-N
    PROP=ANR/ANS
    BB=10.0**S
    A=2.0*BB
    CC=10.0**(-S)
    IF(NNZ-NV)998,998,220
220 AK=0.0
    DO 221 J=1,NV
    AK=AK+X(J)
    IF(X(J))997,300,400
400 U(J)=X(J)
    C(J)=-1.0
    GO TO 221
300 U(J)=999999.9999
    C(J)=-10.0
221 CONTINUE
222 DO 223 J=1,NV
    IF(X(J))224,224,225
224 GO TO 223
225 C(J)=-5.0
    U(J)=2.0* X(J)
    GO TO 900
223 CONTINUE
900 RETURN
997 WRITE(2,8)J,X(J)
  8 FORMAT(23HYOU DUM-DUM, YOU SET X(,I5,11H) EQUAL TO ,E20.7,21H; GEN
   1ERATION ABORTED. )
9999 RETURN
    END


    SUBROUTINE CRE8
    COMMON/TOUGH/ M,N,RHO,S,AK,NNZ,NO,NV,ANR,ANS,PROP
    COMMON/BALLIT X(5000),U(5000),GI(5000),C(5000)
    COMMON/GOOF/A,BB,CC,KDUZ
    DIMENSION CH(2)
    DATA CH(1),CH(2)/1HX,1HS/
  2 FORMAT (13H   FILE         PHD , 67X)
    WRITE   (6, 2)
  1 FORMAT(4X,10HSUBMATRIX: ,66X)
    WRITE(6,1)
    K1=NV+10000
    DO 100 L=1,2
101 FORMAT(4X,10(A1,I5,1X),6X)
100 WRITE(6,101) (CH(L),I,J=10001,K1)
105 FORMAT(4X,6HRHS..1,70X)
    WRITE(6,105)
106 FORMAT(4X,4HR000,72X)
    WRITE(6,106)
107 FORMAT(4X,6(F11.4,1X),4X)
    WRITE(6,107)(C(J),J=1,NV)
    B=0.0
    WRITE(6,107)(B,J=1,NV))
    K1=M+9999
    DO 500 I=10001,K1
108 FORMAT(4X,1HR,I5,70X)
    WRITE (6,108)I
    CALL GGEN(I-10000)
    WRITE(6,107)(GI(J),J=1,NV),(GI(J),J=1,NV)
    B=0.0
    DO 230 J=1,NV
230 B=B+GI(J)*U(J)
112 FORMAT(4X,2H= ,F11.4,61X)
500 WRITE(6,112)B
510 K1TEMP=K1+1
    WRITE(6,108)K1TEMP
    B=1.0
    WRITE(6,107) (B,J=1,NV)
    B=0.0
    WRITE(6,107 (B,J=1,NV)
    WRITE(6,512)AK
512 FORMAT(4X,2H= ,F11.4,10X,1H;,52X)
    WRITE(6,601)
601 FORMAT(4H END,76X/4H EOF,76X)
    ENDFILE 6
    REWIND 6
    RETURN
    END
```

- - - If the matrix is being printed out of rather than read into a matrix generator the "REWIND 6" should be removed. - - -

Received January 1974; revised May 1974