IMPLICIT ENUMERATION AND CONSTRAINT ORDERING
FOR DISCRETE OPTIMIZATION PROBLEMS

By

Walrave T. Jansen
A Thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in Partial fulfillment of the requirements for the degree of Master of Science (Mineral Economics).

Signed: Walgrave T. James
Student

Golden, Colorado
Date: \textit{June 20}, 1975

Approved: \textit{Thesis Advisor}

Golden, Colorado
Date: \textit{June 20}, 1975
ABSTRACT

A great amount of work has been done to solve integer and 0,1 problems. Several algorithms have been developed to come up with optimal solutions by using partial enumeration techniques. The most well known of these are the method of Balas and that of Lawler and Bell.

When a scheduler is deciding which projects to start, given certain constraints, he must decide which of a variety of codes to use to determine a course of action.

Partial enumeration techniques, even though they are much more efficient than total enumeration, are still very slow due to the time consuming nature of continual constraint evaluation and references to subscripted variables.

It is the intent of this paper to show how assembly language techniques may be used to accelerate the solution finding process considerably. Furthermore, other improvements are also considered.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>PROBLEM FORMULATION</td>
<td>3</td>
</tr>
<tr>
<td>THE LAWLER AND BELL SOLUTION ALGORITHM.</td>
<td>4</td>
</tr>
<tr>
<td>ASSEMBLY LANGUAGE UTILIZATION</td>
<td>6</td>
</tr>
<tr>
<td>Description of constraint evaluating function</td>
<td>7</td>
</tr>
<tr>
<td>CONSTRAINT ORDERING ALGORITHM</td>
<td>10</td>
</tr>
<tr>
<td>Generalized constraint tightness formula</td>
<td>13</td>
</tr>
<tr>
<td>Method of obtaining relative constraint tightness</td>
<td>13</td>
</tr>
<tr>
<td>Table 1, Cumulative Binomial Coefficients</td>
<td>17</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>18</td>
</tr>
<tr>
<td>Table 2, Computational Results</td>
<td>20</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>21</td>
</tr>
<tr>
<td>Flow chart of main program</td>
<td>21</td>
</tr>
<tr>
<td>Flow chart of assembly language function</td>
<td>22</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>23</td>
</tr>
<tr>
<td>COMPUTER LISTINGS</td>
<td>Map pocket</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I want to thank the members of my graduate committee, Professor Jean P. Mather and Professor Hunter S. Swanson, for their assistance during the preparation of this thesis. I am especially indebted for the help and guidance provided by the chairman of my graduate committee, Professor Robert E. D. Woolsey.
INTRODUCTION

The types of problems that are most readily solved by 0,1 programs are those that involve the evaluation of various alternative projects that compete for limited resources. The payoff maximization or cost minimization is determined in the objective function, while the resource limitations are represented by the constraints.

Although there are a variety of relatively quick techniques for coming up with good, but not necessarily optimal, solutions to such problems, only a few time consuming methods guarantee the optimum solution. Of the latter the implicit enumeration technique of Egon Balas and that of Lawler and Bell are probably the most well known.

The method described by Egon Balas involves a great deal of housekeeping and maintenance of lists of variables. Consequently, programs using this technique are complicated, large and time consuming.

The Lawler and Bell algorithm is much more readily programmed on a computer. A description of the algorithm
is given in the next section of this thesis.

The thesis will begin by defining the type of problems to be solved. A method for generating and using the solution vector showing the advantages of techniques that are available in assembly language only, is then described. Further techniques for accelerating the search are introduced and finally, computational results are compared and evaluated.
PROBLEM FORMULATION

The object of the Lawler and Bell solution technique through partial enumeration is to solve problems in the following form:

MINIMIZE: \( g_0(x) \),

SUBJECT TO:

\[ g_{11}(x) - g_{12}(x) \geq 0 \]

\[ g_{21}(x) - g_{22}(x) \geq 0 \]

\[ g_{m_1}(x) - g_{m_2}(x) \geq 0 \]

where \( x_j = 0 \) or \( 1 \) for \( j = 1,2,3,...,n \) and \( x = \) the vector \( x_j \), \( j = 1,2,3,...,n \) and the functions \( g_0, g_{11}, g_{12}, g_{21},..., g_{m_2} \) are all monotonically non-decreasing with respect to the \( x_j \)'s.
THE LAWLER AND BELL SOLUTION ALGORITHM

Since the values of the various x's are zero or one, the solution vector x can be considered as a binary number. These can be tested for optimality and feasibility in numerical order from (0,0,0,...,0) to (1,1,1,...,1). If all were examined, $2^n$ different vectors would have to be checked.

The Lawler and Bell algorithm sets up the following three rules which allow one to skip a considerable number of these vectors:

1) If $g_0(x) \geq g_0(\hat{x})$, skip to $x^*$.

2) If x is a feasible solution and $g_0(x) < g_0(\hat{x})$, skip to $x^*$.

3) If, for any $i$ ($i = 1,2,...,m$), $g_{i1}(x^*-1) - g_{i2}(x) < 0$, skip to $x^*$.

where $x^* = (x \text{ OR } (x-1)) + 1$, or $x^* = x$ plus one added to the lowest order bit in x that is a one. The justification for rule one is that the current best solution for the
problem (\(\hat{x}\)) is better than any solution between \(x\) and \(x^*\). This is so since any vectors between \(x\) and \(x^*\) will have one or more bits to the right of the lowest order 'one' bit in \(x\), increasing the value of the objective function even further. Since we are trying to minimize the value of the monotonically non-decreasing objective function, any additional 'one' bits in \(x\) will only worsen the result.

Rule two is justified by the same reasoning.

For rule three a different approach must be taken. If we take a certain \(x\) and check it against a particular constraint which is not satisfied by \(x\), a method is devised to see if turning on all the variables that have a positive coefficient to the right of the lowest order 1 will satisfy the constraint. For this, \(x^*-1\) is generated and the maximum value for the constraint between \(x\) and \(x^*\) is found. If this maximum value still does not satisfy the constraint one should continue from \(x^*\), since the constraint will be unsatisfied in the vector range \(x\) through \(x^*-1\).
ASSEMBLY LANGUAGE UTILIZATION

Since the program spends most of its time checking and evaluating constraints it is important that this section of the program is most efficient.

First we must examine what is involved when we evaluate a constraint. A solution vector $x$, consisting of zeroes and ones representing the value of the variables in the constraint, is used to evaluate the left hand side of the constraint by including those coefficients whose corresponding variable value is one and ignoring the others.

The 'x' vector of variables would be similar to a binary word and can actually be treated as such. In Fortran this is not very practical as the word would have to be broken down into its individual bits every time a constraint is evaluated. This breaking down might be avoided by performing a logical 'or' with a single bit that could be shifted by multiplying by two or with a location in an array that contained consecutive powers of two. Both of these methods would be time consuming since multiplication is a relatively slow instruction and the indirect address-
ing necessary to compare to a changing array location is also inefficient. If it is not contained in a single word, incrementing it, generating \( x^* \) and storing it becomes laborious.

For these reasons, which are caused by the limitations inherent in the Fortran language, it was decided to write an assembly language function that would evaluate the left hand side of the constraints. To improve its performance, the constraint coefficient matrix was set up so that the coefficients for a constraint would be contained in two rows, one for the positive and one for the negative coefficients. Furthermore, each row would consist of the appropriate number of consecutive locations in core. This latter is done by assigning the variable number to the first subscript in the array and the constraint number to the second subscript. By doing this the function can treat each row of coefficients as a singly subscripted variable when given the starting location of the row.

A listing of the function is given at the end of this thesis. It should be borne in mind that such assembly routines are usable only on the computer make and model they are written for, since assembly languages are machine dependent. Many instructions however, are similar to those in other assembly languages, and therefore the listing may still be considered of value to the reader not actually working on a Digital Equipment PDP-10 computer.
A brief description of the method used in the function follows:

The information transferred to the function is the solution vector IX, the beginning location of the row of coefficients in the constraint coefficient matrix called IG, and, the number of variables (NV) in the problem. Note that all the coefficients and the constant should be stored as positive numbers in the IG array.

First the NV is negated and stored in the left half of accumulator N. The row address in the IG array is transferred to the right half of this same accumulator. Next a one is stored in accumulator M and a zero in S. Accumulator S serves simultaneously as the sum of the eligible coefficients and the value returned to the calling program.

Now we go to the instruction identified by LOOP:. This is the beginning location of a loop that will check all the variable values against their corresponding coefficients and add such coefficients that are matched to variables that have a value of one.

A logical 'and' is performed between the IX vector and the M register, if the masked bit is zero, the summing instruction is not performed. If the masked bit is a one, the value of the coefficient stored in the address in the right hand side of the N accumulator is added to the S accumulator. Next the mask in the M accumulator is shifted left one bit.

The next instruction AOBJN N,5 is one that is rather
complex and will need some additional explanation. The
mnemonic literally means, Add One to Both halves of the
accumulator and Jump if the result is Negative. This
means that NV, which is stored in the left half as a neg-
ative number is decremented (i.e., becomes less negative),
while the address in the right half is changed to the next
location in core. If the counter is still negative (i.e.,
not all variables have been considered yet) return to the
location LOOP to check the next variable. When all vari-
ables have been checked, the constant, which should be
stored in the appropriate row representing positive or
negative coefficients immediately following the last var-
iable, is added to the sum. Thereupon the routine returns
to the calling program. Due to the extremely small size
of the loop that accumulates the coefficients, in which
a coefficient may be added for every four machine instruc-
tions executed, a considerable time saving is effected.
It was considered to be of primary importance that
the program get to the constraint that would be unsatis-
fied as soon as possible. To do this the constraints are
sequenced in order of decreasing tightness. This way those
constraints that cause failure most often will be checked
first.

Relative tightness between constraints may be deter-
mined by comparing the number of possible solutions that
would not satisfy the constraints being compared. Due
to the variations in the number of non-zero coefficients in
the constraints, it was found that a more convenient num-
ber than the total failing solutions was the ratio of
failing solutions to the total number of solutions pos-
sible. It is obvious that each additional 0,1 variable
added to a problem increases the number of different
solution possibilities by a factor of two, even when the
coefficient of such a variable is zero in the particular
constraint. If the coefficient is zero however, the value
of that variable has no effect upon the ratio of infeasible
solutions to constraint satisfying solutions. The reason for this is that the value of the constraint when evaluated with a certain solution vector is unaffected by the value of the variable. Therefore, if the constraint is already unsatisfied, it will remain so.

To demonstrate this, and the method for evaluating relative constraint tightness, an example follows.

First a simplification is made so that the mathematics is rigorous. This limits the variable coefficients in all constraints to be either -1, 0 or 1. Many problems are limited in this way so that this assumption is not excessively limiting or unrealistic. For example the Combinatorial and Haldi's IBM problems described by Trout and Woolsey (2). The right hand sides of the constraints and the objective function coefficients may be any positive or negative integer or zero. Negative objective function coefficients are eliminated by substituting the complement of the associated variable throughout the problem.

Taking an arbitrary constraint

\[ x_1 - x_2 + 0x_3 - x_4 - x_5 + 1 \geq 0 \]

we can readily see that it may be satisfied or fail depending upon the 0,1 solution vector \( x \). Note that the right hand side constant of one has been transferred to the left so that this example conforms to the mode of the problem formulation.

If we wanted to determine how many solution vectors
do not satisfy the constraint we could evaluate all 32 different combinations for the five different variables and come up with the number. This could be done quite simply for just five variables since there are only 32 solution vectors, however a 20 variable constraint would give us $2^{20}$ or 1,048,576 solution vectors. Returning to our problem, we find that the minimum value for the constraint is -2. This is generated by the two solution vectors $(0,1,0,1,1)$ and $(0,1,1,1,1)$. We see that the value of $x_3$ has no effect upon the constraint feasibility. Similarly, the maximum value of two is generated by the solutions $(1,0,0,0,0)$ and $(1,0,1,0,0)$. Again the value of $x_3$ is immaterial, showing that the zero coefficient increases the number of solutions but does not change their ratio. $x_3$ will therefore be ignored so that we come up with the resulting constraint:

$$x_1 - x_2 - x_4 - x_5 + 1 \geq 0.$$

To this constraint, there are sixteen different solution vectors ranging in lexicographical order from $(0,0,0,0,0)$ to $(1,1,1,1,1)$. If we were to list all solutions we would find that each variable occurs as a 'one' eight times and as a 'zero' eight times. This suggests that the probability that any variable is either 0 or 1 is one half.

An infeasible solution vector is defined as one that generates a negative value for the left hand side. The
minimum value of \(-2\) can be generated by only one of the sixteen solution vectors, namely \((0,1,1,1)\). The next most negative solution that can be obtained is \(-1\). This can be arrived at in a variety of ways. A general formula will be set up to determine what fraction of the solutions will give any particular value for the left-hand side.

Letting \(p\) be the probability that a variable is 1 and \(q\) the probability that it is a 0 in any single solution vector, and letting all variables be treated as equals, we find that the probability that a particular solution will occur is:

\[
\binom{n}{r} p^r q^{n-r},
\]

(1)

where \(\binom{n}{r}\) is defined as \(\frac{n!}{(n-r)!r!}\), \(n\) is the total number of non-zero coefficients in the constraint and \(r\) is the difference between the number of variables with positive coefficients set to 1 and the number of variables with negative coefficients set to 1 subtracted from the number of negative coefficients.

Let us determine the probability of obtaining \(-1\) as a result of a solution vector in the above constraint using the formula.

\(n\), being the number of non-zero coefficients is equal to four. To arrive at a value of \(-1\) for the value of the constraint, the sum of coefficients must be \(-2\) since the
constant will increase the sum by 1. To obtain a value of -2 for the left hand side, the number of non-zero variables corresponding to negative coefficients will exceed those non-zero variables corresponding to positive coefficients by two. The total number of negative coefficients is three and thus \( r \) becomes \( 3 - 2 = 1 \).

Since the probability of zero or one occurring for a variable in a solution vector is equal, the actual number of zeroes and ones is immaterial and \( p^{r \cdot (n-r)} \) becomes \( \left( \frac{1}{2} \right)^n \).

Now inserting the values of \( n \) and \( r \) into the formula we get

\[
\binom{n}{r} \left( \frac{1}{2} \right)^n = \binom{4}{1} \left( \frac{1}{2} \right) = \frac{4!}{1!3!} \left( \frac{1}{2} \right)^4
\]

\[
= \frac{24}{1 \times 6} \times \frac{1}{16} = \frac{4}{16}.
\]

This means that of the 16 solution vectors, four will generate an infeasible solution of -1 to this constraint. When these are added to the single non-feasible solution of -2 we get a total of five non-feasible solution vectors to the constraint. Therefore, as all other solution vectors will be feasible, the ratio of infeasible to total solutions is 5:16.

If we then generalize the formula so that all infeasible solutions are summed we get:

\[
\sum_{r=0}^{i} \binom{n}{r} \frac{1}{2^n}
\]

where \( i \) is the number of negative coefficients minus the
constant.

To determine what number of possible zero-one solution vectors to a constraint, having \(-1, 0, +1\) coefficients and integer right hand side, will not satisfy the constraint, use the following set of steps:

1) Determine the number of non-zero coefficients in the constraint. (=NNZC)

2) Determine the number of negative coefficients in the constraint. (=NEG C)

3) Subtract the constant in the constraint from NEG C. (=NC)

At this point NC can be used to determine the following:

a) If NC > NNZC the constraint cannot be satisfied.

b) If NC = NNZC the constraint can only be satisfied by setting the variables corresponding to positive coefficients equal to one and those corresponding to negative coefficients equal to zero.

c) If NC ≤ 0 the constraint cannot be infeasible and may be eliminated from the problem.

At this point the fraction of infeasible solutions can be calculated using the generalized formula (2), but a quicker method can be employed by using Table 1, which represents the accumulation of the binomial coefficients by row. It shows the number of unsatisfied solution variables given NNZC and NC. This number should then be divided by \(2^{\text{NNZC}}\) to determine the infeasible fraction.
The degree to which these infeasible fractions overlap and what ranges of solution vectors can be eliminated from consideration would be a good topic for further consideration.
CONCLUSION

The use of assembly language programming is effective when used in programs that solve zero-one problems. Solution vectors are represented as binary words and the various operations to be performed on them are readily programmed using assembly language instructions. The Lawler and Bell algorithm is particularly adaptable to this technique.

Constraint ordering is very important when there are many constraints in the problem and can cut program execution time by a significant fraction. It should be noted however that execution time is much more sensitive to the number of variables than to the number of constraints. In a twenty variable - twenty constraint problem an additional constraint may add at most five per cent to execution time and may in fact reduce execution time; an additional variable may increase execution time by up to one hundred per cent.

Currently the program can only handle problems with thirty-five variables or less. This is due to the limitations of the PDP-10 machine which has a thirty-six bit
word length. To expand the program would be possible, but would make it less efficient. If the program is expanded care must be taken that no problem is attempted that will require total enumeration. A 30 x 30 problem that has to be fully enumerated would take approximately one week of CPU time on a PDP-10 computer. A 40 x 40 problem would take about twenty-five years of CPU time.

Table 2 shows the solution times for the Assembly Language Programming System (ALPS) versus the various programs mentioned in reference 2 (Trauth and Woolsey). The program is apparently much faster for those problems with few constraints (the Allocation and Haldi's IBM problems), while still doing fairly well on the simpler Combinatorial problems. It should be noted that for Haldi's problem number nine, an improved objective function value of eight was obtained rather than nine, as reported in the article.
<table>
<thead>
<tr>
<th>Problem</th>
<th>T-1761</th>
<th>LP1</th>
<th>ILP2-1</th>
<th>ILP2-2</th>
<th>IPSSC</th>
<th>ALPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.77</td>
<td>14</td>
<td>2.42</td>
<td>19</td>
<td>1.94</td>
<td>44</td>
</tr>
<tr>
<td>2</td>
<td>4.08</td>
<td>31</td>
<td>5.08</td>
<td>55</td>
<td>3.87</td>
<td>63</td>
</tr>
<tr>
<td>3</td>
<td>4.03</td>
<td>30</td>
<td>3.90</td>
<td>41</td>
<td>3.77</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>3.68</td>
<td>18</td>
<td>2.60</td>
<td>19</td>
<td>4.19</td>
<td>192</td>
</tr>
<tr>
<td>5</td>
<td>3.50</td>
<td>11</td>
<td>1.97</td>
<td>12</td>
<td>4.19</td>
<td>139</td>
</tr>
<tr>
<td>6</td>
<td>3.73</td>
<td>18</td>
<td>3.87</td>
<td>40</td>
<td>3.45</td>
<td>157</td>
</tr>
<tr>
<td>7</td>
<td>4.73</td>
<td>61</td>
<td>7.88</td>
<td>81</td>
<td>2.91</td>
<td>119</td>
</tr>
<tr>
<td>8</td>
<td>3.83</td>
<td>21</td>
<td>4.41</td>
<td>51</td>
<td>3.63</td>
<td>370</td>
</tr>
<tr>
<td>9</td>
<td>3.60</td>
<td>12</td>
<td>1.87</td>
<td>12</td>
<td>4.13</td>
<td>201</td>
</tr>
</tbody>
</table>

**Table 2: Computational Results**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.93 2.4 11.67</td>
</tr>
<tr>
<td>2</td>
<td>633.31 6758 473.10</td>
</tr>
</tbody>
</table>

Time in seconds
APPENDIX A

START

INPUT CONSTRAINT MATRIX AND INITIALIZE VARIABLES

X \rightarrow X + 1

HAVE ALL SOLUTIONS BEEN CHECKED?

YES

NO

\text{g}_0(X) \geq g_0(X)?)

YES

NO

\text{g}_{i1}(x^* - 1) \leq g_{i2}(x) \hspace{1cm} (i = 1, 2, \ldots, m)

YES

NO

\text{g}_{i1}(x) \leq g_{i2}(x) \hspace{1cm} (i = 1, 2, \ldots, m)

YES

NO

\text{STOP}

X \rightarrow X^*

OUTPUT RESULTS

\hat{X} \rightarrow x
This function \( g(x) \) does the following:

1. Tests for those variables that have a value of '1' in the current solution vector 'IX'.
2. If a variable is found to be '1' add the corresponding coefficient in the constraint matrix to the sum.
3. Finally, the constant is added to the sum if its sign is the same as the coefficients currently being summed.
4. The function returns to the calling program with the sum of the coefficients whose corresponding variables are equal to '1' plus the constant if applicable.
BIBLIOGRAPHY


PROBLEM IDENTIFICATION

IDENTIFY FUNCTION INSTEP AS AN EXTERNAL ASSEMBLY-LANGUAGE
SUBPROGRAM.

EXTERNAL IS0FP.

SUBROUTINE IS0FP.

SET UP INPUT OUTPUT DEVICES.

INPUT.

INPUT THE FUNCTION IDENTIFICATION.

INPUT THE NUMBER OF SOLUTIONS.

INPUT THE NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE OBJECTIVE FUNCTION COEFFICIENTS.

INPUT THE CONSTANT COEFFICIENTS.

INPUT THE FAILURE CONDITIONS.

INPUT THE MAXIMUM NUMBER OF ITERATIONS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.

INPUT THE MAXIMUM NUMBER OF CONSTRAINTS.

INPUT THE MAXIMUM NUMBER OF OBJECTIVE FUNCTION VALUES.

INPUT THE MAXIMUM NUMBER OF VARIABLE VALUES.
MULTIPLES A BINARY VECTOR IN WORD IX BY A ROW OF COEFFICIENTS IN ARRAY IG AND SUMS THE RESULTS.

ENTRY: IGOFX

银

HITLE: IGOFX

NOYX X(0) MOVE X TO LEFT HALF OF WORD

MOR X,1 GET BEGINNING ADDRESS OF IG ROW

DOES 1,1 PUT A ONE IN N

HRELZ 4,2 LOOP MOVE ADDRESS OF LOOP TO LEFT HALF

BLT 4,12 TRANSFER LOOP TO REGISTERS

/ THE NEXT SECTION IS TRANSFERRED TO REGISTERS N=11 BY JR.

LOOP: DNE X,10 IF MASKED BIT IS ZERO

ADD M,N ADD WORD IN IG ARRAY TO SUM

M,(0) SHIFT MASK LEFT ONE PLACE

ADJS N,(0) DECREMENT COUNTER, INCREMENT LOC.

ADD M,(N) ADD CONSTANT

JRA 0,30 RETURN TO CALLING PROGRAM