THE COMPUTER SOLUTION OF PROBLEMS IN INTEGER PROGRAMMING

M.R. GUY.

Ph.D. Thesis September, 1969.

ABSTRACT.

The thesis is concerned largely with Gomory's Method of Integer Forms whereby an integer programming problem is solved by a combination of linear programming operations and the addition of new constraints.

Chapter 1 describes the theory behind the method. It deals with the techniques of linear programming when the use of floating point and its associated rounding and truncation errors are avoided and describes the way in which new constraints can be generated and added during solution of the problem.

Chapter 2 deals with the author's experiments in integer programming. Parts 1 to 3 are concerned with the linear programming method which was developed partly to deal with numerical problems and partly to facilitate the choice of constraints. Part 4 deals with experiments with different criteria for choosing constraints.

Chapter 3 is concerned with two algorithms. The first is essentially the lexicographic method advocated by Haldi and Isaacson. An independent approach has provided an insight into it which led to the development of the second algorithm. In this the objective function is replaced by approximations to it with smaller coefficients in order to obtain an approximate solution more rapidly. A restriction is then placed on the objective function and a search made for a better solution.

Chapter 4 compares the two algorithms of Chapter 3 with those of certain other authors. It is concluded that the systematic method of choosing constraints used in the author's algorithms enables them to be regarded as special forms firstly of a branch and bound algorithm and secondly of a backtrack method. As a corollary it is suggested that some of the techniques used in the author's algorithms to speed up solution could be applied to these other methods.

ACKNOWLEDGEMENTS.

I am indebted to the Science Research Council for awarding me a Research Studentship to enable me to study at the University of Newcastle upon Tyne from October 1964 to September 1967. Also to Professor E.S. Page for admitting me to the Computing Laboratory to research for a Ph.D.

My thanks are due to Dr H.I. Scoins for his supervision during my three years at Newcastle, and after, and for his ideas and comments from which I have benefitted.

I must thank Wiggins Teape Research and Development Ltd. for their encouragement and for arranging to type this thesis.

Lastly, but not least, I have to thank my wife for enduring a thesis that has occupied many evenings and weekends.

TABLE OF CONTENTS.

Abstract		2
Acknowledgements		3
Index to Figures Contained in the Text.		6
Chapter 1:	Background	8
Part 1:	The Simplex Method	8
Part 2:	The Matrix Algebra of Linear Programming	12
Part 3:	Extending the Problem during Computation	18
Part 4:	The Integer Properties of the Rows of the Transformed Matrix	20
Part 5:	The Generation of Additional Constraints	28
Part 6:	The Lexicographic Dual Simplex Method	39
Part 7:	The Use of Integer Arithmetic within a Digital Computer	42
Part 8:	The Euclidean Algorithm	44
Chapter 2:	Experiments in Integer Programming	47
Part 1:	Linear Programming as a Subset of Integer Programming	48
Part 2:	The Linear Programming Structure of the Experimental Programmes	51
Part 3:	Summary of the Structure of the Experimental Programmes	61
Part 4:	Description and Comparison of the Experimental Programmes	64
Chapter 3:	The Two Most Effective Algorithms	78
Part 1:	The Significance of a Lexicographic Method	79
Part 2:	The Dependance of the Rate of Convergence upon the Ordering of the Basic Variables in the Tableau	82
Part 3:	The Problem of the Dual Function of the Cost Row	84
Part 4:	A Numerical Illustration of the Use of Artificial Cost Functions	87
Part 5:	Aspects of the Algorithm which would Benefit from Further Research	93
Chapter 4:	Comparison of the Methods Described in this Thesis with the Work of Other Authors	98
Part 1:	Haldi and Isaacson (ref. 9)	98
Part 2:	Martin (ref. 10)	99
Part 3:	Land and Doig (ref. 12)	101
Part 4:	Backtrack Methods	103

References:		106
Appendix A:	Symbols, Notations and Definitions	109
Appendix B:	The Test Data	113
Part 1:	Description of the Problems	113
Part 2:	The Problems	124
Appendix C:	Tables giving the Results of Running the Experimental Programmes on the Test Data	135
Appendix D:	The Experimental Programmes	168
Part 1:	The Two Most Effective Programmes	168
Part 2:	The Other Programmes	194

INDEX TO FIGURES CONTAINED IN THE TEXT

Figure 1.5.1:	Example of Derivable Cuts	37
Figure 2.2.1:	An Example of Scaling	55
Figure 2.2.2:	An Example of Looping in Integer Programming	58
Figure 2.3.1:	An Outline Flowchart of Programme BHD	63
Figure 2.4.1:	An Illustration of the Methods Described in Part 4 of Chapter 2	73
Figure 3.4.1:	A Comparison of Three Different Ways of Solving a Simple Problem	89

CHAPTER 1

BACKGROUND

Chapter 1 : Background

Part 1 : The Simplex Method

The major part of this thesis is concerned with a method of integer programming the basis of which was laid in a paper by R.E. Gomory: An algorithm for Integer Solutions to Linear Programs (ref. 1). It is concerned with the solution of the linear programming problem:

$$\text{Minimise} \quad \sum_{j=1}^{n} c_j x_j$$

subject to the constraints

$$\sum_{j=1}^{n} w_{ij} x_j \lesseqgtr b_i \qquad (i=1\ldots m)$$

$$x_j \geq 0 \qquad (j=1\ldots n)$$

where the w_{ij}, b_i, c_j are constants,

the x_j are variables

and $\lesseqgtr$ means $\leq$ or $=$ or $\geq$,

and the additional integer constraints:

x_j integer.

In our treatment of the problem we will assume that all the constraints, apart from the ones $x_j \geq 0$, are in the form $\sum w_{ij} x_j = b_i$. This does not involve loss of generality for the inequalities $\sum w_{ij} x_j \leq b_i$ and $\sum w_{ij} x_j \geq b_i$ can be expressed as the pairs $\sum w_{ij} x_j + s_i = b_i$, $s_i \geq 0$ and $\sum w_{ij} x_j - s_i = b_i$, $s_i \geq 0$, respectively. Also we will normally use matrix notation.

Accordingly we may restate the problem as:

minimise $\underline{c}' \, \underline{z}$ 1.1.1.

subject to $W\underline{z} = \underline{b}$

and $\underline{z} \geq 0$

where A is an m x n matrix, $\underline{b}$ is an m-dimensional vector, and $\underline{c}$ and $\underline{z}$ are n-dimensional vectors. It will be convenient for us to partition W into $[B, N]$, where B is a non-singular square matrix of order m, and N is an m x(n-m) matrix. Partitioning $\underline{c}'$ into $[\underline{c}_1', \underline{c}_2']$ and $\underline{z}'$ into $[\underline{x}', \underline{y}']$ we have:

$$\text{minimise} \quad \underline{c}_1' \underline{x} + \underline{c}_2' \underline{y} \qquad 1.1.2.$$

$$\text{subject to} \quad B\underline{x} + N\underline{y} = \underline{b}$$

$$\underline{x}, \underline{y} \geq 0$$

This system can be solved for $\underline{x}$ in terms of $\underline{y}$:

$$\underline{x} = B^{-1}\underline{b} - B^{-1}N\underline{y} \qquad 1.1.3.$$

and the problem becomes

$$\text{minimise} \quad \underline{c}_1' B^{-1}\underline{b} + (\underline{c}_2' - \underline{c}_1' B^{-1}N)\underline{y} \qquad 1.1.4.$$

$$\text{subject to} \quad B^{-1}\underline{b} - B^{-1}N\underline{y} \geq 0, \ \underline{y} \geq 0.$$

The basic result of linear programming theory is that $\underline{y} = 0$, $\underline{x} = B^{-1}\underline{b}$ will cause $\underline{c}_1' B^{-1}\underline{b}$ to be a minimum solution provided $B^{-1}\underline{b} \geq 0$ and $\underline{c}_2' - \underline{c}_1' B^{-1} N \geq 0$. If one or both of these conditions are not satisfied an element of $\underline{x}$ and an element of $\underline{y}$ are selected according to certain rules and exchanged. It is not considered necessary to prove these results here and the rules for selecting an x and a y are simply stated with references.

Let us denote by $(B)_{ij}$ the ijth element of matrix B, and by $(\underline{c})_i$ or $(\underline{c}')_i$ the ith element of vector $\underline{c}$.

If $B^{-1}\underline{b} \geq 0$ but $\underline{c}_2' - \underline{c}_1' B^{-1}N \not\geq 0$ we select an element $(\underline{y})_j$ of $\underline{y}$ such that its coefficient $(\underline{c}_2' - \underline{c}_1' B^{-1}N)_j$ in the objective function is < 0 (usually the most negative). It is then exchanged with an $(\underline{x})_i$ where i is such that $(B^{-1}N)_{ij} > 0$, and $(B^{-1}\underline{b})_i/(B^{-1}N)_{ij}$ is a minimum over all i. This ensures that $B^{-1}\underline{b}$ remains ≥ 0. This procedure is known as the Simplex method (ref. 3, p.70).

On the other hand if $\underline{c}_2' - \underline{c}_1' B^{-1}N \geq 0$ but $B^{-1}\underline{b} \ngeq 0$ we select an element $(\underline{x})_i$ of $\underline{x}$ such that $(B^{-1}\underline{b})_i < 0$ (usually the most negative) and then exchange it with $(\underline{y})_j$ where j is such that $(B^{-1}N)_{ij} < 0$ and $-(\underline{c}_2' - \underline{c}_1' B^{-1}N)_j/(B^{-1}N)_{ij}$ is a minimum over all j. This ensures that $\underline{c}_2' - \underline{c}_1' B^{-1}N$ remains ≥ 0. This procedure is known as the Dual Simplex method (ref. 3, p.99).

If both $B^{-1}\underline{b} < 0$ and $\underline{c}_2' - \underline{c}_1' B^{-1}N < 0$ a "composite" method is used. The use of composite methods is not yet given much space in linear programming textbooks. The general principle is to construct some function of the infeasibilities and non-optimal cost elements, and choose pivots to make this function tend toward zero. One such is a "self-dual parametric algorithm" (ref. 4, p.245).

These three methods are used once a non-singular matrix, B, has been found, as in equation 1.1.3. Such a matrix can be obtained by inventing an artificial vector, $\underline{v}$, to form a basis and then eliminating it. For example, in order to find a matrix, B, in the problem 1.1.2. one would start by solving the sub-problem:

$$\begin{aligned} &\text{minimise} && \underline{v}' \\ &\text{subject to} && I\underline{v} + B\underline{x} + N\underline{y} = \underline{b} \\ & && \underline{v}, \underline{x}, \underline{y} \geq 0. \end{aligned}$$

Providing a non-singular matrix, B, exists the minimum of $\underline{v}$ is zero, and once this minimum has been obtained $\underline{v}$ can be left out of the equations and the problem takes the form 1.1.4.

The introduction of $\underline{v}$ presents us with a unit matrix, I, as a starting point. Sometimes B will contain some unit vectors, when slack variables have been used to turn inequalities into the equalities of equations 1.1.2. In such a case as many of these unit vectors as possible will be used to make up the initial unit matrix, an artificial vector being used to fill the remaining columns.

If $\underline{b}$ contains negative elements when the problem is expressed in the form 1.1.1. one has two kinds of infeasibility in the same problem.

One composite method of solving this is described by Wolfe in (ref. 6). It chooses pivots which will reduce a function of the infeasibilities until the problem becomes feasible. Thereafter the normal form of the Simplex method is used to obtain an optimal solution.

The programmes contained in Appendix D were designed in such a way that problems entered two procedures, first one that performed the ordinary Simplex method for both eliminating artificial variables and obtaining an optimal solution, and secondly one that carried out the Dual Simplex method. The intention of this was to be able to accept problems in either of two forms. However it was found to work successfully on problems which were infeasible in both ways and also non-optimal. It is not suggested that this method was efficient.

Part 2: The Matrix Algebra of Linear Programming

The purpose of this Part is to establish the algebra of pivoting used in linear programming in a slightly different form to that normally used. This is in order that we might know at any stage of computation exactly which quantities are integer and which are not.

The problem is:

$$\text{Minimise} \quad \underline{c}_1' \, \underline{x} + \underline{c}_2' \, \underline{y} \qquad 1.2.1.$$

$$\text{subject to} \quad B \, \underline{x} + N \, \underline{y} = \underline{b}$$

$$\underline{x} \geq 0, \ \underline{y} \geq 0 \text{ and } \underline{x}, \ \underline{y} \text{ integer.}$$

We have separated the variables into basic variables, $\underline{x}$, and non-basic variables, $\underline{y}$. If the original inequalities are written in the form

$$\underline{x} + B^{-1}N \, \underline{y} = B^{-1}\underline{b} \qquad 1.2.2.$$

we are assuming that the rows and columns have been suitably ordered and we may consider it either as an expression for $\underline{x}$ in terms of $\underline{y}$ when viewed as

$$\underline{x} = B^{-1}\underline{b} - B^{-1}N \, \underline{y}$$

or as a collection of entries in a table which is to be manipulated whilst working towards the desired optimal solution. We shall refer to this table as the tableau.

Since we are essentially working with integers throughout this thesis all matrices, vectors and scalars will be taken to be integer unless otherwise stated. In particular all coefficients in the original problem, i.e. $\underline{c}_1$, $\underline{c}_2$, B, N, and $\underline{b}$ will be integer.

Wherever possible the use of inverse matrices will be avoided and adjugate matrices used instead. Let us write d for $|B|$, the

determinant of B, and B* for the adjugate matrix of B, so that $dB^{-1} = B^*$ and $BB^* = B^*B = dI$. Since $(B^*)_{ij}$ is the cofactor of $(B)_{ij}$ in B, the elements of B* are integers, (ref. 5, p.87) which is of course why we prefer the use of B* to that of B^{-1}. Accordingly we will normally express 1.2.2. in the form

$$d\underline{x} + B^*N\underline{y} = B^*\underline{b} \qquad 1.2.3.$$

The letters d and D will be used exclusively for $|B|$. Gomory, in (ref. 1), uses D and this will be followed in some parts of this thesis, notably the appendices. In this chapter d will be used to emphasise the fact that it is a scalar.

Before proceeding with the main part of this section we need to establish two preliminary results.

Lemma 1.2.1. Let a non-singular square matrix B be partitioned by its last row and column into $\begin{bmatrix} B_1 & \underline{h}_1 \\ \underline{h}_2' & h_3 \end{bmatrix}$, where B_1 is also non-singular. Then $|B|$, the determinant of B, may be written as

$$|B| = h_3 \; |B_1| - \underline{h}_2' \, B_1^* \, \underline{h}_1 \qquad 1.2.4.$$

To show this we add to the last column of the partitioned form of B the vector formed by postmultiplying the previous columns by $- B_1^{-1} \underline{h}_1$. Thus

$$|B| = \begin{vmatrix} B_1 & 0 \\ \underline{h}_2' & h_3 - \underline{h}_2' B_1^{-1} \underline{h}_1 \end{vmatrix} = |B_1| \; (h_3 - \underline{h}_2' B_1^{-1} \underline{h}_1)$$

$$= h_3 \; |B_1| - \underline{h}_2' \, B_1^* \, \underline{h}_1$$

since $|B_1| \, B_1^{-1} = B^*$.

Lemma 1.2.2. Let B be defined and partitioned as in Lemma 1.2.1. Then B* may be written in the partitioned form

$$B^* = \begin{bmatrix} \dfrac{|B|\ B^*_1}{|B_1|} + \dfrac{B^*_1 \underline{h}_1 \underline{h}'_2 B^*_1}{|B_1|} & - B^*_1 \underline{h}_1 \\ - \underline{h}'_2 B^*_1 & |B_1| \end{bmatrix} \quad 1.2.5.$$

To show this we observe that

$$B_1^{-1} = \begin{bmatrix} B_1^{-1} + \gamma B_1^{-1} \underline{h}_1 \underline{h}'_2 B_1^{-1} & - \gamma B_1^{-1} \underline{h}_1 \\ -\gamma \underline{h}'_2 B_1^{-1} & \gamma \end{bmatrix} \quad 1.2.6.$$

where $\gamma\ (h_3 - \underline{h}'_2 B_1^{-1} \underline{h}_1) = 1$,
as can be verified by pre- or post-multiplication by the partitioned form of B. As shown in Lemma 1, $|B| = |B_1|\ (h_3 - \underline{h}'_2 B_1^{-1} \underline{h}_1)$, and we may write this in the form $\gamma\ |B| = |B_1|$. Substituting this expression for γ in 1.2.6. and multiplying by $|B|$ we obtain 1.2.5.

We now examine the process of 'pivoting' the expression 1.2.3. This implies choosing a particular element of $\underline{x}$ together with a particular element of $\underline{y}$, exchanging the elements between the two vectors and reforming the relevant matrices. Let us suppose that the expression 1.2.3. becomes

$$\bar{d}\ \bar{\underline{x}} + \bar{B}^*\bar{N}\bar{\underline{y}} = \bar{B}^*\underline{b}$$

after one pivot, $\bar{\underline{x}}$ and $\bar{\underline{y}}$ each differ from $\underline{x}$ and $\underline{y}$ in exactly one element, and $\bar{B}$ and $\bar{N}$ each differ from B and N in exactly one column. $\bar{d}$ is defined to be $|\bar{B}|$.

The pivot element of the transformation of B*N into $\bar{B}^*\bar{N}$ is defined to be that element of B*N contained in the row corresponding to the chosen element in $\underline{x}$ and the column corresponding to the chosen element of $\underline{y}$. We assert that $\bar{d}$ is equal to this pivot element.

For convenience let us assume that it is the last element of $\underline{x}$ and the last element of $\underline{y}$ that are to be exchanged. Then the pivot element will be in the last row and column of B*N. We partition B and N by the last row and column:

$$B = \begin{bmatrix} B_1 & \underline{h}_1 \\ \underline{h}_2' & h_3 \end{bmatrix}, \quad N = \begin{bmatrix} N_1 & \underline{n}_1 \\ \underline{n}_2' & n_3 \end{bmatrix}$$

Then $\bar{B}$ and $\bar{N}$ will take the form

$$\bar{B} = \begin{bmatrix} B_1 & \underline{n}_1 \\ \underline{h}_2' & n_3 \end{bmatrix} \quad \bar{N} = \begin{bmatrix} N_1 & \underline{h}_1 \\ \underline{n}_2' & h_3 \end{bmatrix},$$

having exchanged the last columns of B and N.

The pivot element in B*N is the inner product of the last row of B* and the last column of N, that is $\begin{bmatrix} -\underline{h}_2' B_1^*, & |B_1| \end{bmatrix} \cdot \begin{bmatrix} \underline{n}_1 \\ n_3 \end{bmatrix}$, making use of the expression for B* proved in lemma 1.2.2. This expands into $|B_1|\, n_3 - \underline{h}_2' B_1^* \underline{n}_1$ which is precisely the determinant of $\bar{B}$.

Some programmes written to perform linear programming hold B*, N and $\underline{b}$ separately and update them at every pivot operation. Others, including those described in this thesis, hold these quantities in combined form, namely B*N and B*$\underline{b}$, and it is these which are updated every pivot operation. Let us write A = B*N and B*$\underline{b}$ = $\underline{p}$ so that 1.2.3. becomes

$$d\underline{x} + A\underline{y} = \underline{p} \qquad 1.2.7.$$

Suppose that A is partitioned into

$$\begin{bmatrix} A_1 & \underline{a}_1 \\ \underline{a}_2' & a_3 \end{bmatrix} \text{ and } \underline{p} \text{ into } \begin{bmatrix} \underline{p}_1 \\ p_3 \end{bmatrix} \qquad 1.2.8.$$

We will show that when a_3 is the pivot element the new array to replace A is given by

$$\bar{A} = \begin{bmatrix} \dfrac{\bar{d}\, A_1 - \underline{a}_1\, \underline{a}_2'}{d} & -\underline{a}_1 \\ \underline{a}_2' & d \end{bmatrix} \qquad \text{where } d = |B| \qquad \text{and} \quad \bar{d} = |\bar{B}| = a_3 \qquad 1.2.9.$$

As $\bar{A} = \bar{B}^*\bar{N}$, the product of two integer matrices, it must be integer itself, and this permits us to deduce that

$$\bar{d}\, A \equiv \underline{a}_1\, \underline{a}_2' \qquad (\text{mod } d) \qquad 1.2.10.$$

This property forms the basis of the discussion in Part 4.

To prove equation 1.2.9. we evaluate the product of B*N using the partitioned form for B* derived in lemma 1.2.2. We have

$$\begin{aligned} a_3 &= -\underline{h}_2'\, B_1^*\, \underline{n}_1 + |B_1|\, n_3 \\ &= \bar{d} \end{aligned}$$

as already shown.

$$\begin{aligned} \underline{a}_2' &= -\underline{h}_2'\, B_1^* N_1 + |B_1|\, \underline{n}_2' \\ \underline{a}_1 &= (|B|\, B_1^*\, \underline{n}_1 + B_1^*\, \underline{h}_1\, \underline{h}_2'\, B_1^*\, \underline{n}_1)/\,|B_1| - n_3\, B_1^*\, \underline{h}_1 \\ &= (|B|\, B_1^*\, \underline{n}_1 + B_1^*\, \underline{h}_1\, (\underline{h}_2'\, B_1^*\, \underline{n}_1 - |B_1|\, n_3))/|B_1| \\ &= (d\, B_1^*\, \underline{n}_1 - \bar{d}\, B_1^*\, \underline{h}_1)/|B_1| \\ A_1 &= (|B|\, B_1^* N_1 + B_1^*\, \underline{h}_1\, \underline{h}_2'\, B_1^* N_1)/|B_1| - B_1^*\, \underline{h}_1\, \underline{n}_2' \\ &= (|B|\, B_1^* N_1 + B_1^*\, \underline{h}_1\, (\underline{h}_2'\, B_1^* N_1 - |B_1|\, \underline{n}_2'))/|B_1| \\ &= (d\, B_1^* N_1 - B_1^*\, \underline{h}_1\, \underline{a}_2')/|B_1| \end{aligned}$$

To obtain the corresponding values for the elements of $\bar{A}$ we simply exchange the values of $\underline{h}_1$ and $\underline{n}_1$, h_3 and n_3, and d and $\bar{d}$. Thus

$$\bar{a}_3 = d$$

$$\bar{\underline{a}}_2' = -\underline{h}_2' B^*_1 N_1 + |B_1| \underline{n}_2'$$

$$\bar{\underline{a}}_1 = (\bar{d} B^*_1 \underline{h}_1 - d B^*_1 \underline{n}_1)/|B_1|$$

$$\bar{A}_1 = (\bar{d} B^*_1 N_1 - B^*_1 \underline{n}_1 \underline{a}_2')/|B_1|$$

We immediately note that $\bar{a}_3 = d$, $\bar{\underline{a}}_2' = \underline{a}_2'$, and $\bar{\underline{a}}_1 = -\underline{a}_1$. To prove the expression for $\bar{A}_1$ we evaluate it:

$$(\bar{d} A_1 - \underline{a}_1 \underline{a}_2')/d = (\bar{d} d B^*_1 N_1 - \bar{d} B^*_1 \underline{h}_1 \underline{a}_2' - d B^*_1 \underline{n}_1 \underline{a}_2' + \bar{d} B^*_1 \underline{h}_1 \underline{a}_2')/d\ |B_1|$$

$$= (\bar{d} d B^*_1 N_1 - d B^*_1 \underline{n}_1 \underline{a}_2')/d\ |B_1|$$

$$= (\bar{d} B^*_1 N_1 - B^*_1 \underline{n}_1 \underline{a}_2')/|B_1|$$

$$= \bar{A}_1$$

It is necessary to comment on the sign of d and $\bar{d}$. The algebra presented so far is valid whether they are negative or positive. However in the remainder of this thesis it will be convenient to assume that d and $\bar{d}$ are positive. Accordingly we adopt the convention that the partitioned form of A in 1.2.8 will transform by pivoting on a_3 into 1.2.9. only when $a_3 > 0$. If $a_3 < 0$ we will define $\bar{d} = -a_3$ and will write $\bar{A}$ as the negative of 1.2.9:

$$\bar{A} = \begin{bmatrix} \dfrac{\bar{d} A_1 + \underline{a}_1 \underline{a}_2'}{d} & \underline{a}_1 \\ -\underline{a}_2' & -d \end{bmatrix} \qquad 1.2.11.$$

Part 3: Extending the Problem during Computation

The method of (ref. 1) consists of two basic steps. The first is pivoting, the algebra for which is described in Part 2, and the second consists of adding new constraints. In order that computation may continue to be performed in integers it is necessary that any new constraint must have integer coefficients in its representation in the original space. To be precise, the original equations can be expressed as

$$B\underline{x} + N\underline{y} = \underline{b} \qquad 1.3.1.$$

and any additional constraint as

$$\underline{k}'\,\underline{x} + s + \underline{n}_2'\,\underline{y} = b_3. \qquad 1.3.2.$$

s can be regarded either as a slack variable, constrained to be non-negative, or an artificial variable constrained to be zero in any feasible solution.

Additional constraints may be added either to the original constraints, in which case it is easy to ensure that the variables are integers, or to the transformed array. In the latter case the transformed array is that obtained by multiplying 1.3.1. through by B*, viz

$$d\underline{x} + B^*N\underline{y} = B^*\underline{b} \qquad 1.3.3.$$

and substituting for $\underline{x}$ in 1.3.2., which gives us, after multiplying 1.3.2. through by d and rearranging,

$$ds + (d\underline{n}_2' - \underline{k}'\,B^*N)\underline{y} = db_3 - \underline{k}'\,B^*\underline{b} \qquad 1.3.4.$$

Thus the coefficient of $\underline{y}$ in the constraint added to the transformed array is the sum of an integer vector each of whose elements is a multiple of d and an integer combination of the coefficients of $\underline{y}$ in the existing constraints. As there is a one to one correspondence between equations 1.3.2. and 1.3.4. it will be seen that the condition that an additional constraint is in the form 1.3.4. is necessary and sufficient for the equivalent constraint added to 1.3.1. to have integer coefficients.

We now consider the possibility of adding a new variable. Although no programme was written which actually did this it facilitates a proof in Part 4. We add a new variable, t, with coefficient $\underline{q}$ to 1.3.1.:

$$B\underline{x} + N\underline{y} + \underline{q}t = \underline{b} \tag{1.3.5.}$$

Multiplying through by B* we obtain

$$d\underline{x} + B^*N\underline{y} + B^*\underline{q}t = B^*\underline{b}. \tag{1.3.6.}$$

If now we restrict $\underline{q}$ to be of the form $B\underline{r}_1 + N\underline{r}_2$ equation 1.3.6. takes the form

$$d\underline{x} + B^*N\underline{y} + (d\underline{r}_1 + B^*N\underline{r}_2)t = B^*\underline{b}$$

which is of a form analogous to 1.3.4.

In Part 4 we shall append a new row and column at the same time so that the tableau of B*N will expand to become

$$\begin{bmatrix} B^*N & B^*N\underline{v} \\ \underline{u}'B^*N & \underline{u}'B^*N\underline{v} \end{bmatrix}$$

Part 4: The Integer Properties of the Rows of the Transformed Matrix.

Gomory has shown that, in general, if we take the rows of matrix B*N modulo d, they generate an additive group of order d. As before, we write A for B*N and d for the determinant of B. We shall prove this result for the predominant case, i.e. when A has no common factor.

Firstly, however, let us consider two examples. The first is from Gomory (ref. 1, p. 297).

Minimise $-3x_1 + x_2$

Subject to

$$3x_1 - 2x_2 + x_3 = 3$$
$$-5x_1 - 4x_2 + x_4 = -10$$
$$2x_1 + x_2 + x_5 = 5$$

Rewriting this in the form of a tableau and optimising we have, indicating the pivot elements by asterisks:

	1	x_1	x_2	
z	0	-3	1	
x_3	3	3*	-2	(d = 1)
x_4	-10	-5	-4	
x_5	5	2	1	

		x_3	x_2	
z	3	1	-1	
x_1	1	1/3	-2/3	(d = 3)
x_4	-5	5/3	-22/3	
x_5	3	-2/3	7/3*	

	1	x_3	x_5	
z	30/7	5/7	3/7	
x_1	13/7	1/7	2/7	(d = 7)
x_4	31/7	-3/7	22/7	
x_2	9/7	-2/7	3/7	

Let us construct a row from the cost row by taking the elements of the cost row modulo d. This is (2, 5, 3). If we now construct the rows obtained by taking successive multiples of this we obtain the sequence

1. (2, 5, 3)
2. (4, 3, 6)
3. (6, 1, 2)
4. (1, 6, 5)
5. (3, 4, 1)
6. (5, 2, 4)
7. (0, 0, 0)

It is not surprising that we obtain exactly 7 distinct rows. What is interesting is that if we now take the rows corresponding to x_1, x_4 and x_2 modulo d we obtain

x_1 (6, 1, 2)

x_4 (3, 4, 1)

x_2 (2, 5, 3)

These are the same as the rows obtained by taking multiples 3, 5 and 1 of the cost row.

This result is a perfectly general one. It may be summed up by saying that the rows generated by taking the rows of the matrix modulo d form an additive group of order d. In other words there

are exactly d such rows and any linear combination of the rows is also a member of the group. In this case the group is cyclic, i.e. there is a member of the group such that every other member of the group may be generated by taking successive multiples of it.

In the next example the group is not cyclic.

$$\text{Minimise} \quad -x_1 - x_3$$

$$\text{Subject to} \quad 2x_1 + 3x_2 + 4x_3 + x_4 = 5$$

$$4x_1 + 3x_2 + 2x_3 + x_5 = 5$$

In tableau form:

	1	x_1	x_2	x_3	
z	0	-1	0	-1	
x_4	5	2	3	4	(d = 1)
x_5	5	4*	3	2	

	1	x_5	x_2	x_3	
z	5/4	1/4	3/4	-1/2	
x_4	5/2	-1/2	3/2	3*	(d = 4)
x_1	5/4	1/4	3/4	1/2	

	1	x_5	x_2	x_4	
z	5/3	1/6	1	1/6	
x_3	5/6	-1/6	1/2	1/3	(d = 12)
x_1	5/6	1/3	1/2	-1/6	

To construct the group of rows modulo d from this we need two rows. For example take the row generated from the cost row, that is (8, 2, 0, 2) and from 3 times the x_3 row, that is (6, 6, 6, 0). We construct 6 rows by taking multiples of the (8, 2, 0, 2) row, and the other 6 by adding the (6, 6, 6, 0) row on to each.

	(a)	(b)
1.	(8, 2, 0, 2)	(2, 8, 6, 2)
2.	(4, 4, 0, 4)	(10, 10, 6, 4)
3.	(0, 6, 0, 6)	(6, 0, 6, 6)
4.	(8, 8, 0, 8)	(2, 2, 6, 8)
5.	(4, 10, 0, 10)	(10, 4, 6, 10)
6.	(0, 0, 0, 0)	(6, 6, 6, 0)

That none of these rows can generate the whole group can be easily demonstrated. If we take successive multiples of any row the 6th element will be the row (0, 0, 0, 0) because every element is a multiple of 2. Thereafter the cycle will repeat.

The rows corresponding to x_3 and x_1 are (10, 10, 6, 4) and (10, 4, 6, 10) and these appear in the second column of the list as numbers 2 and 5.

Before we prove the main result we must establish a preliminary one. This is that if the elements of a matrix W have a highest common factor of g, there exist vectors $\underline{u}$, $\underline{v}$, such that

$$\underline{u}' W \underline{v} \equiv g \pmod{h}$$

where h is any given integer.

To illustrate this consider the following matrix, whose elements are shown in factorised form:

$$\begin{bmatrix} 2.\ 3.\ 5.\ 7.\ 11 & 2.\ 3.\ 5.\ 7.\ 13 & 2.\ 3.\ 5.\ 11.\ 13 \\ 2.\ 3.\ 7.\ 11.\ 13 & 2.\ 5.\ 7.\ 11.\ 13 & 3.\ 5.\ 7.\ 11.\ 13 \end{bmatrix} \quad 1.4.1.$$

and suppose that h = 2. 3. 5. 7. 11. 13.

The highest common factor of the elements of this matrix is 1 and so we must choose vectors with which to pre- and post- multiply the matrix so that the resultant scalar has no factors in common with d. This is easily done, for if we premultiply the matrix by [1,1] and post-multiply by [1,1,1]' this effectively sums the elements of the matrix and this sum is prime to h. For example, the sum will not

have 5 as a factor for it may be written as

$$2.\ 3.\ 7.\ 11.\ 13. + 5(2.\ 3.\ 7.\ 11. + \ldots\ldots + 3.\ 7.\ 11.\ 13).$$

To further the example, suppose now that $h = 2.\ 3.\ 5.\ 7.\ 11.\ 13.\ 17.$ We cannot easily determine whether the sum of the elements of matrix 1.4.1. is a multiple of 17 or not, but this is not necessary if we change the pre- and post- multiplying vectors to $[1,17]$ and $[1,17,17]'$.

The point of this is to show that we needed to determine only five variables, that is the elements of $\underline{u}$ and $\underline{v}$, to generate a number congruent to g whereas if the six elements of the matrix had been arranged as a vector, $\underline{w}$, any vector $\underline{t}$ such that $\underline{t}'\underline{w} \equiv 1 \pmod d$ would have to have six non-zero elements.

The following lemma makes use of two arguments in particular. Firstly; that given a vector $\underline{w}$, there is a vector $\underline{t}$ such that $\underline{t}'\underline{w}$ is equal to the highest common factor of $\underline{w}$. Secondly; that if r is prime to s, r + as is prime to s.

Lemma 1.4.1. Let W be an integer matrix whose elements have a highest common factor g, and let h be any integer. Then there exist integer vectors $\underline{u}$, $\underline{v}$ such that

$$\underline{u}'\ W\ \underline{v} \equiv g \pmod h \qquad 1.4.2.$$

To show this we first write $W = gT$, and rewrite equation 1.4.2. as

$$\underline{u}'\ T\ \underline{v} \equiv 1 \pmod h$$

To establish this result let h be expressed as the product of its prime factors:

$$h = \prod_{i=1}^{n} p_i^{\ q_i}.$$

For each index i we select a row which does not contain p_i as a common factor. Denote this row by k (i). We take this row which we denote as before by $T_{k(i)*}$ and multiply it by every other factor of h:

$$T_{k(i)*} \prod_{j \neq i} p_j^{q_j}.$$

This still does not contain p_i as a common factor, but does have as a factor every p_j where $j \neq i$.

Sum this expression over i. As every one except the i th contains p_i as a common factor the resultant vector will not contain p_i as a common factor. The sum can be expressed as

$$\underline{u}' T$$

where

$$\underline{u} = \sum_i \underline{e}_{k(i)} \prod_{j \neq i} p_j^{q_j}$$

where $\underline{e}_{k(i)}$ has 1 as its k(i) th element and zero elsewhere.

As $\underline{u}'$ T does not contain any p_i as a common factor there exists a vector $\underline{v}$ such that

$$\underline{u}' T \underline{v} \equiv 1 \pmod{h}.$$

The vector $\underline{v}$ can be generated by means of the Euclidean Algorithm.

We now set out to prove the result illustrated at the beginning of this chapter, namely that when the matrix A = B*N has no common factor its rows taken modulo d generate a group of order d, where d = $|B|$.

It is assumed initially that A is of the form B*N where B and N are integer matrices, and that d = $|B|$. It is not assumed at this point that A has no common factor.

Let us choose vectors $\underline{u}$ and $\underline{v}$ such that $\underline{u}'$ A $\underline{v}$ is the highest common factor of A. As established in Part 3, A can be extended by adding a row and a column to become

$$\begin{bmatrix} A & A\underline{v} \\ \underline{u}'A & \underline{u}'A\underline{v} \end{bmatrix}$$

This matrix has the same properties as A in that it represents the product of two integer matrices, one of which is the adjugate of an integer matrix with determinant d. Accordingly we may use the property established in Part 2 (equation 1.2.10) that

$$(\underline{u}' A \underline{v}) A \equiv (A \underline{v})(\underline{u}' A) \pmod{d}$$

Let us write g for $\underline{u}' A \underline{v}$

$$g A \equiv A \underline{v} \underline{u}' A \pmod{d} \qquad 1.4.3.$$

We wish to establish a property concerning row vectors generated by taking an integer linear combination of the rows of A. Let us denote a typical row vector as $\underline{w}' A$, where $\underline{w}$ is any m-dimensional vector. Multiplying both sides of 1.4.3. by $\underline{w}'$ we obtain

$$g \underline{w}' A \equiv \underline{w}' A \underline{v} \underline{u}' A \pmod{d}$$

We now assume that $g = 1$. If we write $\mu = \underline{w}' A \underline{v}$ we have

$$\underline{w}' A \equiv \mu \underline{u}' A \pmod{d}$$

As $\underline{u}'$ is fixed independently of $\underline{w}$, and μ can have at most d distinct values modulo d this shows that $\underline{w}' A$ can have at most d distinct values modulo d.

To show that $\underline{w}' A$ can in fact take on d distinct values we first observe that μ can take on d distinct values. For let $\underline{w}' = \lambda \underline{u}'$, where $\lambda = 1, .., d$. Then

$$\mu = \underline{w}' A \underline{v} = \lambda \underline{u}' A \underline{v} = \lambda$$

Secondly the d values of μ will generate d distinct values of $\mu \underline{u}' A$. For if not suppose

$$\mu_i \underline{u}' A \equiv \mu_j \underline{u}' A \text{ where } \mu_i \not\equiv \mu_j$$

Then postmultiplying each side of the equation by $\underline{v}$ we obtain

$$\mu_i \underline{u}' A \underline{v} \equiv \mu_j \underline{u}' A \underline{v} \pmod{d}$$

But we have assumed that $\underline{u}' A \underline{v} = 1$.

Therefore $\quad \mu_i \equiv \mu_j \quad$ contrary to hypothesis.

Therefore $\underline{w}' A$ can take on d distinct values.

This result is a very general one, and fails only when A has a common factor. The proof of lemma 1.4.1. is a constructive one and shows us how to construct the group of permissible rows. For each of the prime factors p_i of d we select a row that contains an element prime to p_i and multiply this row by $d/p_i^{\,q_i}$, where q_i is the number of times p_i is repeated as a factor of d. The sum of these will generate the whole group.

In practice of course if an element exists which is prime to d the row that contains it will generate the whole group of constraints. Experimentation showed that the group was usually cyclic, and also could often be generated by a single row of A.

Part 5: The Generation of Additional Constraints

Methods of integer programming in general need to augment the constraints contained in the original statement of the problem with constraints derived during the process of solution. In this part we consider four ways of deriving such constraints in such a way as to fill the following conditions

(a) the constraints must have integer coefficients when represented in the original space.

(b) they must not render infeasible any feasible integer point.

(c) they must exclude from the feasible space some part of it which contains no integer point.

To satisfy the first condition we write any new constraint as

$$\underline{k}' \, \underline{x} + s + \underline{n}_2' \, \underline{y} = b_3 \qquad 1.5.1.$$

where $\underline{k}$, $\underline{n}_2$ and b_3 are integer constants, and s is an integer variable. This is its representation in the original space, that is when added to the equation 1.3.1. Recalling Part 3 we remember that 1.5.1., then referred to as 1.3.2., takes on the form 1.3.4. when added to the transformed set of equations 1.3.3. We reproduce 1.3.4. here as 1.5.2.

$$d\, s + (d\, \underline{n}_2' - \underline{k}' \, B^*N)\underline{y} = d\, b_3 - \underline{k}' \, B^*\underline{b} \qquad 1.5.2.$$

The rest of the section describes different approaches to defining the values of $\underline{k}$, $\underline{n}_2$ and b_3.

First however we introduce a new pair of symbols [ ]. We define them to be such that $[a]$ represents the largest integer not greater than a. We also extend the definition so that $[a]_d$ represents the largest multiple of d not greater than a. We can define the second usage in terms of the first:

$$[a]_d = d\left[\frac{a}{d}\right].$$

We will make use of the relations

$$a - \left[a\right]_d \geq 0 \qquad 1.5.3.$$

and

$$a - \left[a\right]_d < d. \qquad 1.5.4.$$

5 (i) Gomory's Original Derivation

This is the derivation presented in (ref. 1).

We have to choose values for the $\underline{k}$, $\underline{n}_2$, and b_3 in 1.5.2. We allow ourselves a choice of $\underline{k}$, subject to conditions discussed later. We then define $\underline{n}_2$ and b_3 such that

$$d\ \underline{n}_2' = \left[\underline{k}'\ B^*N\right]_d \text{ and } d\ b_3 = \left[\underline{k}'\ B^*\underline{b}\right]_d. \qquad 1.5.5.$$

Substituting in 1.5.2. we obtain

$$ds = -\left\{\left[\underline{k}'\ B^*N\right]_d - \underline{k}'\ B^*N\right\}\underline{y} + \left\{\left[\underline{k}'\ B^*\underline{b}\right]_d - \underline{k}'\ B^*\underline{b}\right\} \quad 1.5.6.$$

and relations 1.5.3. and 1.5.4. enable us to deduce that

$$ds > -d.$$

As s is integer this implies $\quad s \geq 0.$

This turns equation 1.5.6., which is simply a definition of s, into an inequality which excludes no integer point. It will exclude some part of the space not containing an integer point provided

$$\left[\underline{k}'\ B^*\underline{b}\right]_d - \underline{k}'\ B^*\underline{b} < 0$$

i.e. $\quad \underline{k}'\ B^*\underline{b} \not\equiv 0 \pmod{d}.$

Attempts to implement the method of (ref. 1) tend to choose $\underline{k}$ so that 1.5.6. looks like a good constraint. For example the constant term might be large and negative, that is approaching the value of $-d$, or the coefficient of $\underline{y}$ might have small elements. The next section describes a derivation based on a different approach.

5(ii) A more direct approach

Dr. Land (ref 7) suggests that good constraints tend to be generated from the sum of small multiples of the original constraints. So whereas others might choose $\underline{k}$ so that $\underline{k}'\ B^*N$ or $\underline{k}'\ B^*\ \underline{b}$ satisfy certain conditions, Dr. Land would choose $\underline{k}$ so that the vector $\underline{k}'\ B^*$ contained small elements.

So we pre-multiply the original equations

$$B\underline{x} + N\underline{y} = \underline{b}$$

by $\underline{k}'\ B^*$ to get

$$d\ \underline{k}'\ \underline{x} + \underline{k}'\ B^*\ N\underline{y} = \underline{k}'\ B^*\ \underline{b}$$

Relation 1,5,3 permits us to write

$$d\ \underline{k}'\ \underline{x} + \left[\underline{k}'\ B^*N\right]_d \underline{y} \leq \underline{k}'\ B^*\underline{b}$$

and since the left hand side is a multiple of d we can round down the right hand side to be a multiple of d also:

$$d\ \underline{k}'\ \underline{x} + \left[\underline{k}'\ B^*N\right]_d \underline{y} \leq \left[\underline{k}'\ B^*\underline{b}\right]_d$$

If we now write ds for the integer slack variable and substitue $B^*\underline{b} - B^*N\underline{y}$ for $d\ \underline{x}$ we obtain

$$ds = \left[\underline{k}'\ B^*\underline{b}\right]_d - \underline{k}'\ B^*\underline{b} - \left(\left[\underline{k}'\ B^*N\right]_d - \underline{k}'\ B^*N\right) \underline{y}$$

as before.

5(iii) A more general approach

This was given by Gomory in (ref 2) and in addition to producing the constraints already described produces the constraints used in his all-integer algorithm contained in the same reference.

Let λ be any non-negative number, not necessarily integer. Define $\underline{r}_1$, $\underline{r}_2$ and r_0 as follows:

$$d\,\underline{k}' = \left[\frac{d\,\underline{k}'}{\lambda}\right]\lambda + \underline{r}'_1 \qquad \text{where } 0 \leq (\underline{r}_1)_i < \lambda, \text{ all } i$$

$$\underline{k}'\,B^*N = \left[\frac{\underline{k}'\,B^*N}{\lambda}\right]\lambda + \underline{r}'_2 \qquad \text{where } 0 \leq (\underline{r}_2)_i < \lambda, \text{ all } i$$

$$\underline{k}'\,B^*\underline{b} = \left[\frac{\underline{k}'B^*\underline{b}}{\lambda}\right]\lambda + r_0 \qquad \text{where } 0 \leq r_0 < \lambda. \qquad 1.5.7.$$

Consider a linear combination of the equations, thus

$$d\,\underline{k}'\,\underline{x} + \underline{k}'\,B^*N\,\underline{y} = \underline{k}'\,B^*\underline{b}$$

where $\underline{k}'$ integer. We may rewrite it using equations 1.5.7. and rearranging the terms:

$$\underline{r}'_2\,\underline{y} + \underline{r}'_1\,\underline{x} = r_0 + \lambda\left\{\left[\frac{\underline{k}'\,B^*\underline{b}}{\lambda}\right] - \left[\frac{\underline{k}'\,B^*N}{\lambda}\right]\underline{y} - \left[\frac{d\,\underline{k}'}{\lambda}\right]\underline{x}\right\} \qquad 1.5.8.$$

Now let us define s to be the contents of the curly bracket:

$$s = \left[\frac{\underline{k}'\,B^*\underline{b}}{\lambda}\right] - \left[\frac{\underline{k}'\,B^*N}{\lambda}\right]\underline{y} - \left[\frac{d\,\underline{k}'}{\lambda}\right]\underline{x} \qquad 1.5.9.$$

Clearly s is integer. But from 1.5.8. we have

$$\lambda s = -r_0 + \underline{r}'_2\,\underline{y} + \underline{r}'_1\,\underline{x} \geq -r_0 > -\lambda$$

since $\underline{r}'_2\,\underline{y} + \underline{r}'_1\,\underline{x} \geq 0$ and $r_0 < \lambda$ from 1.5.7.

As s is integer, we deduce

$$s \geq 0.$$

Now let $\lambda = d$ and define $\underline{n}_2$ and b_3 as before (1.5.5.). Then 1.5.9. becomes

$$s = \left[\frac{\underline{k}' \; B^*\underline{b}}{d}\right] - \left[\frac{\underline{k}' \; B^*N}{d}\right] \underline{y} - \underline{k}' \; \underline{x}$$

On multiplying through by d and substituting $B^*\underline{b} - B^*N\underline{y}$ for $d \; \underline{x}$ we again obtain

$$ds = \left[\underline{k}' \; B^*\underline{b}\right]_d - \underline{k}' \; B^*\underline{b} - \left\{\left[\underline{k}' \; B^*N\right]_d - \underline{k}' \; B^*N\right\} \underline{y} \; . \qquad 1.5.10.$$

In the all-integer algorithm Gomory derived constraints in such a way that the pivot element was always -1 and hence $d \equiv 1$.
These constraints were derived by making $\lambda > d$ in 1.5.9. and adding that constraint to the transformed tableau. The value of λ was chosen large enough to ensure a pivot of -1.

The method was designed to work with a tableau in dual feasible form and $\underline{k}$ was chosen so that $\underline{k}' \; B^*\underline{b} < 0$, thus ensuring that $\left[\underline{k}' \; B^*\underline{b}/\lambda\right] < 0$.

5(iv) Gomory's mixed integer method

In (ref 8) Gomory gives a method of deriving constraints where some but not all of the variables must take on integer values. It is of interest here because it does not reduce to the method of Part 5(i) when there are no non-integer variables present.

We consider an integer combination of the transformed tableau:

$$d\,\underline{k}'\,\underline{x} + \underline{k}'\,B^*N\,\underline{y} = \underline{k}'\,B^*\underline{b} \qquad 1.5.11.$$

We restrict the choice of $\underline{k}$ to ensure that $\underline{k}'\,\underline{x}$ is a combination of integer variables. To make the algebra more readable let us write x for $\underline{k}'\underline{x}$ and $\underline{a}_1' - \underline{a}_2'$ for $\underline{k}'\,B^*N$ where $\underline{a}_1 \geq 0$ and $\underline{a}_2 \geq 0$, that is to say every element of $\underline{a}_1$ and $\underline{a}_2$ is greater or equal to zero. 1.5.11. then becomes

$$d\,x + (\underline{a}_1' - \underline{a}_2')\,\underline{y} = \underline{k}'\,B^*\underline{b} \qquad 1.5.12.$$

We assume that although x must be integer valued $\underline{k}'\,B^*\underline{b}$ is not a multiple of d at this point.

We define $f = \underline{k}'\,B^*\underline{b} - \left[\underline{k}'\,B^*\underline{b}\right]_d.$

We have two alternatives.

Either (a) $d\ x \leq \underline{k}'\ B^*\underline{b} - f$ 1.5.13.

or (b) $d\ x \geq \underline{k}'\ B^*\underline{b} - f + d$ 1.5.14.

Suppose (a) is true. Eliminating d x between 1.5.12. and 1.5.13. we have

$$- (\underline{a}_1' - \underline{a}_2')\ \underline{y} \leq - f$$

Since $d - f > 0$ we have

$$- (d - f)(\underline{a}_1' - \underline{a}_2')\ \underline{y} \leq - (d - f)\ f.$$

Since $- d\ \underline{a}_2'\ \underline{y} \leq 0$ we may add it to the left hand side:

$$- d\ \underline{a}_2'\ \underline{y} - (d - f)\ \underline{a}_1'\ \underline{y} + (d - f)\ \underline{a}_2'\ \underline{y} \leq - (d - f)\ f$$

or

$$- (d - f)\ \underline{a}_1'\ \underline{y} - f\ \underline{a}_2'\ \underline{y} \leq - (d - f)\ f. \qquad 1.5.15.$$

On the other hand suppose (b) is true.

Then 1.5.12. and 1.5.14. give us

$$- (\underline{a}_1' - \underline{a}_2')\ \underline{y} \geq - f + d.$$

Since $- f < 0$:

$$f\ \underline{a}_1'\ \underline{y} - f\ \underline{a}_2'\ \underline{y} \leq - f(d - f)$$

and since $- d\ \underline{a}_1'\ \underline{y} \leq 0$:

$$- (d - f)\ \underline{a}_1'\ \underline{y} - f\ \underline{a}_2'\ \underline{y} \leq - f(d - f). \qquad 1.5.16.$$

So either 1.5.15. is true or 1.5.16. is true. But we have so arranged them as to be exactly the same and thus may add the constraint

$$d\ s - ((d - f)\ \underline{a}_1' + f\ \underline{a}_2')\ \underline{y} = - f(d - f) \qquad 1.5.17.$$

where $s \geq 0$.

Rearranging 1.5.17. slightly and resorting to the definitions of $\underline{a}_1 - \underline{a}_2$ and f we may write 1.5.17. as

$$d\ s + (f\ \underline{k}'\ B^*N - d\ \underline{a}_1')\ \underline{y} = f\ \underline{k}'\ B^*\underline{b} - f\ ([\underline{k}'\ B^*\underline{b}]_d + d)$$

which is of the form necessary to ensure that it has integer coefficients in the original space, as may be seen by comparing it with 1.3.4.

In general, s is not an integer variable for its value depends on the non-integer variable $\underline{y}$. However if all the variables are integer s will be too. Furthermore we may strengthen 1.5.17. Instead of 1.5.12. we write

$$d\ x + (\underline{a}_1 - [\underline{a}_1]_d - (\underline{a}_2 - [\underline{a}_2]_d))\ \underline{y} \equiv \underline{k}'\ B^*b \pmod d. \qquad 1.5.18.$$

Relations 1.5.13. and 1.5.14. remain the same and an analogous argument results in a constraint at least as strong as 1.5.17.

$$d\,s - ((d - f)(\underline{a}_1' - [\underline{a}_1']_d) + f\,(\underline{a}_2' - [\underline{a}_2'])) \,\underline{y} = - f(d - f). \qquad 1.5.19.$$

There is no longer any point in distinguishing between $\underline{a}_1$ and $\underline{a}_2$ If we denote the coefficient of a typical element of $\underline{y}$ as a_i we may replace it by $a_i - n\,d$ in any equation of a similar form to 1.5.18. The i th coefficient of $\underline{y}$ in 1.5.19. will be least negative, thus making the constraint strongest, if a_i is replaced by $a_i - [a_i]_d$ giving a coefficient in 1.5.19. of

$$- (d - f)\,(a_i - [a_i]_d) \qquad 1.5.20$$

or by $a_i - [a_i]_d - d$ giving a coefficient of

$$f\,(a_i - [a_i]_d - d). \qquad 1.5.21.$$

1.5.20. will be less negative than 1.5.21 if and only if

$$a_i - [a_i]_d < f.$$

It should be remarked that in (ref. 8) Gomory writes 1.5.16 as

$$- \underline{a}_1' \,\underline{y} - \frac{f}{d - f}\,\underline{a}_2' \,\underline{y} \leq - f.$$

Although this would appear to be better scaled than 1.5.16 it does not usually represent an equation with integer coefficients.

Figure 1.5.1: Examples of derivable cuts

Example taken from Gomory (ref. 1).

Minimise $z = 3x_1 - x_2$

Subject to
$$3x_1 - 2x_2 + x_3 = 3$$
$$-5x_1 - 4x_2 + x_4 = 10$$
$$2x_1 + x_2 + x_5 = 5$$

In full tableau form :

	1	x_1	x_2	x_3	x_4	x_5
z	0	3	-1	0	0	0
x_3	3	3*	-2	1	0	0
x_4	-10	-5	-4	0	1	0
x_5	5	2	1	0	0	1

After 1st pivot:

	1	x_1	x_2	x_3	x_4	x_5
z	3	0	-1	1	0	0
x_1	1	1	-2/3	1/3	0	0
x_4	-5	0	-22/3	5/3	1	0
x_5	3	0	7/3*	-2/3	0	1

Optimal:

	1	x_1	x_2	x_3	x_4	x_5
z	30/7	0	0	5/7	0	3/7
x_1	13/7	1	0	1/7	0	2/7
x_4	31/7	0	0	-3/7	1	22/7
x_2	9/7	0	1	-2/7	0	3/7

Inverse matrix:

1	5/7	0	3/7
0	1/7	0	2/7
0	-3/7	1	22/7
0	-2/7	0	3/7

Set of constraints derived by 6(i),(ii) and (iii):

Added to optimal tableau

$$x_3 + 2x_5 \geq 6$$
$$2x_3 + 4x_5 \geq 5$$
$$3x_3 + 6x_5 \geq 4$$
$$4x_3 + x_5 \geq 3$$
$$5x_3 + 3x_5 \geq 2$$
$$6x_3 + 5x_5 \geq 1$$

Added to original tableau

$$x_1 \leq 1$$
$$2x_1 \leq 3$$
$$3x_1 \leq 5$$
$$2x_1 - x_2 \leq 2$$
$$3x_1 - x_2 \leq 4$$
$$4x_1 - x_2 \leq 6$$

Figure 1.5.1. continued

Additional constraints derived by 5(iv)

Corresponding constraint derived by 6(i)	New constraint added to optimal	New constraint added to original
$x_3 + 2x_5 \geq 6$	$x_3 + 2x_5 \geq 6$	$x_1 \leq 1$
$2x_3 + 4x_5 \geq 5$	$4x_3 + 8x_5 \geq 10$	$4x_1 \leq 6$
$3x_3 + 6x_5 \geq 4$	$9x_3 + 4x_5 \geq 12$	$5x_1 - 2x_2 \leq 5$
$4x_3 + x_5 \geq 3$	$9x_3 + 4x_5 \geq 12$	$5x_1 - 2x_2 \leq 5$
$5x_3 + 3x_5 \geq 2$	$4x_3 + 8x_5 \geq 10$	$4x_1 \leq 6$
$6x_3 + 5x_5 \geq 1$	$x_3 + 2x_5 \geq 6$	$x_1 \leq 1$

Part 6: The lexicographic dual simplex method

In Part 1 we described the dual simplex method. We select an $(\underline{x})_i$ such that $(B^{-1}\underline{b})_i < 0$ and exchange it for a $(\underline{y})_j$ (i.e. pivot on $(B^{-1}N)_{ij}$) where $_j$ is such that $(B^{-1}N)_{ij} < 0$ and

$$- (\underline{c}_2' - \underline{c}_1' B^{-1}N)_j/(B^{-1}N)_{ij} \qquad 1.6.1.$$

is a minimum over j. Unfortunately if there is more than one j for which this is a minimum one needs some criterion to choose between them, and it has been shown that a bad choice can, theoretically at least, give rise to looping, i.e. returning to the same basis again and again until the programme is thrown off (ref. 3, pp. 84, 104).

One of the ways of combating this is the lexicographic method. The algorithm is simply that if for a given set of j the ratio 1.6.1. has the same value, we examine the ratios

$$- (B^{-1}N)_{1j}/(B^{-1}N)_{ij}$$

for the same j, and choose the column j which gives the minimum ratio. If there is another tie, the process is repeated for the next row, and so on.

The finiteness of this process is easily demonstrated using the algebra of Part 2. We assume that the tableau of equation 1.2.7, partitioned as in 1.2.8, is lexicographically optimal, i.e. not only is the first element in each column ≥ 0, but the first non-zero element in every column is > 0. We say that each column is lexicographically positive.

Let us also assume that a_3 has been chosen as pivot, so that $p_3 < 0$, $a_3 < 0$, and when the column $-\underline{a}_1 / a_3$ is compared with analogous columns, i.e. columns $-(A_1)_{*j}/(\underline{a}_2')_j$ such that $(\underline{a}_2')_j < 0$, the first element which differs from the corresponding element of the other column is less than it. We use $(A_1)_{*j}$ to denote the jth column of A_1.

The partitioned tableau of A in 1.2.8. transforms into 1.2.11, since we are considering the case when $a_3 < 0$.

We show, firstly, that once the tableau is lexicographically optimal it will remain so. Choose a typical column of A , let it be the jth. It will transform into $(\bar{d}(A_1)_{*j} + \underline{a}_1 (\underline{a}'_2)_j)/d$.

We must show that the first non-zero element of this column is > 0. If $(\underline{a}'_2)_j > 0$ we are summing two columns which are lexicographically positive, and so the result will be lexicographically positive also.

If $(\underline{a}'_2)_j < 0$, we make use of the fact that the pivot column was chosen because

$$-\underline{a}_1 / a_3 < -(A_1)_{*j} / (\underline{a}'_2)_j ,$$

where the 'less than' sign is used in the sense of 'lexicographically less than'. Writing $\bar{d}$ for $- a_3$ and rearranging, this relation becomes

$$\bar{d}\ (A_1)_{*j} + \underline{a}_1\ (\underline{a}'_2)_j > 0$$

as we wished to show.

Secondly we must show that the tableau never repeats itself. Consider the right hand side of equation 1.2.7, partitioned as in 1.2.8, divided by d, as it would be held in normal linear programming. $\underline{p}_1 / d$ will transform into $\underline{p}_1 / d - p_3\ \underline{a}_1 / d\ a_3$. As $p_3 < 0$, $a_3 < 0$, $d > 0$, and the first non-zero element of $\underline{a}_1$ is > 0, the first non-zero element of $-p_3\ \underline{a}_1 / d\ a_3$ is < 0. Thus the first element of $\underline{p}_1 / d$ to change at any pivot step decreases, and so the tableau can never go back on itself.

This does not mean the same solution can never occur more than once. In general the pivot element will not be chosen from the last row as in 1.2.8. and it is possible for the order of the basic variables to be permuted. For example the solution $x_1 = 3$, $x_2 = 2$, could appear a second time as $x_2 = 2$, $x_1 = 3$.
For this reason the programmes described in this thesis used a stronger form of the lexicographic method.

The equation

$$d\ x + B^*N\ \underline{y} = B^*\underline{b} \qquad 1.6.2.$$

was extended to

$$d \begin{bmatrix} \underline{x} \\ \underline{y} \end{bmatrix} + \begin{bmatrix} B^*N \\ -dI \end{bmatrix} \underline{t} = \begin{bmatrix} B^*\underline{b} \\ \underline{0} \end{bmatrix} \qquad 1.6.3.$$

A pivot row is first copied to the bottom of the tableau, and after pivoting is discarded. The same goes for any new constraint. By this means the elements of $\underline{t}$ will change but not the elements of $\underline{x}$ and $\underline{y}$. Equation 1.6.2. can always be reconstructed by eliminating $\underline{t}$ from equation 1.6.3. This is the system adopted in the example in (ref. 2, p. 204). In the examples in (ref. 1, p.295) new constraints are added to a tableau of the form 1.6.2., and constraints can only be discarded when this can be done without discarding an element of $\underline{x}$ or $\underline{y}$, i.e. when the basic variable is not an element of $\underline{x}$ or $\underline{y}$. This system has the advantage over the other in that 1.6.2. is easier to store than 1.6.3., but this advantage is lost when the equations are held is sparse form, i.e. only non-zero elements are stored.

In general the elements of $\underline{x}$ and $\underline{y}$ in 1.6.3. are interspersed.

Part 7: The use of integer arithmetic within a digital computer

The simplest arithmetic operations of a digital computer are addition, subtraction, multiplication and division of integers. These are usually quicker than other forms of arithmetic and invariably give the exact answer. There is just one proviso - that the answer must not be too big for the location which is to hold it. When this happens the programmer either has to resort to double or multiple length integers, or use fixed point decimals or floating point numbers.

For many people, the advantages of floating point and decimal arithmetic outweigh the perils, but one would hope that in integer programming, of all subjects, one would be able to use the computer for what it is best at. In an all-integer method (ref.2) every number in the constraint matrix is an integer, and if any are so big as to cause overflow, the use of floating point will not solve the problem. This is because floating point can only be used when the answer is only required to within a certain percentage. In integer programming the answer is required to the nearest integer, irrespective of what percentage accuracy this represents.

In the method we are discussing the coefficients are assumed to start off as integers but do not remain so during the calculation. Our treatment of the method has been designed to show how they can be held as integers with a common denominator of d. In deriving new constraints, e.g. 1.5.6., the new coefficients are of the form

$$\left[\underline{a}'\right]_d - \underline{a}' \qquad 1.7.1.$$

or

$$\left[\frac{\underline{a}'}{d}\right] - \frac{\underline{a}'}{d} \qquad 1.7.2.$$

which is the form they would take if floating point were used. If the value of $\left[\frac{\underline{a}'}{d}\right]$ were of the order of ten one decimal place would be lost in accuracy. If $\left[\frac{\underline{a}'}{d}\right]$ were very big much more would be lost. When we consider that multiples of $\underline{a}'$ may be taken, e.g.

$$\left[\left[\frac{d}{2}\right]\frac{\underline{a}'}{d}\right] - \left[\frac{d}{2}\right]\frac{\underline{a}'}{d} \qquad 1.7.3.$$

half the accuracy of $\underline{a}'$ / d could be lost in one operation if d were sufficiently big.

The disadvantage of using integer arithmetic would seem to be the limit on the size of $\underline{a}$ and d. But this is rather doubtful. For if $\underline{a}$ were floating point and allowed to exceed the limit it would cease to be accurate to the nearest integer and any hope of generating constraints from large multiples would be lost. If d lost its accuracy we would not even know how many constraints could be generated.

Because it was desired to experiment with large multiples of constraints without having to worry about accuracy, the programmes were written to hold the coefficients as an integer array with common denominator d. Tests were made for overflow and when this happened a constraint of the type described in Part 5(i) was added. This always leads to a pivot < d which is in fact the new d, and this leads to an overall reduction in the size of the elements of the array A. The transformations

$$\left[\mu\, \underline{a}' \right]_d - \mu\, \underline{a}'$$

and

$$\frac{\bar{d}A_1 - \underline{a}_1 \underline{a}'_2}{d}$$

were performed in double length.

In many problems solved there was no danger of overflow, and only occasionally was great difficulty experienced because of it. But this may have been because the examples tried were mostly simple compared with the potential problems of integer programming.

It was interesting that the experiments suggested that complicated multiples and combinations of constraints did not justify the effort needed (see Chapter 2). If only small multiples were used it might be feasible to use floating point arithmetic. However it is likely that accuracy would be better preserved by deriving a $\underline{k}'$ B^* and calculating the new constraint direct from the original ones in the manner of Part 5(ii) of this chapter.

Part 8: The Euclidean Algorithm

The programmes in Appendix D include a procedure to perform the Euclidean Algorithm. This part is a verification of it.

The integer procedure euclidalg (h, D) computes an integer w such that

$$w\,h \equiv hcf\,(h, D) \mod (D)$$

where hcf (h, D) denotes the highest common factor of h and D. The value of w is assigned to euclidalg.

The procedure initiates four variables:

$$h_o = h - \left[h\right]_D, \quad k_o = D, \quad u_o = 1, \; v_o = 0$$

and iterates as follows

$$k_{r+1} = k_r - h_r \left[k_r / h_r\right], \; v_{r+1} = v_r - u_r \left[k_r / h_r\right]$$

alternately with

$$h_{r+1} = h_r - k_{r+1} \left[h_r / k_{r+1}\right], \; u_{r+1} = u_r - v_{r+1} \left[h_r / k_{r+1}\right]$$

and stops as soon as h_{r+1} or k_{r+1} becomes zero. If k_{r+1} is the first to become zero h_r is the highest common factor of h_o and k_o, if h_{r+1} is the first to become zero k_{r+1} is the required number. At this point w is given by u_r if k_{r+1} is zero and $(u_{r+1} + v_{r+1})$ if h_{r+1} is zero.

Firstly we verify the formula for the highest common factor. We observe that since

$$k_{r+1} = k_r - h_r \left[k_r / h_r\right]$$

any number that divides k_r and h_r also divides k_{r+1}. Conversely any number that divides k_{r+1} and h_r also divides k_r and so

$$hcf\,(h_r, k_r) = hcf\,(h_r, k_{r+1}).$$

Similarly we can show that

$$hcf\,(h_r, k_{r+1}) = hcf\,(h_{r+1}, k_{r+1})$$

Thus by induction we have

$$hcf\,(h_o, k_o) = hcf\,(h_r, k_{r+1}) = hcf\,(h_{r+1}, k_{r+1})$$

which will be h_r if $k_{r+1} = 0$ and k_{r+1} if $h_{r+1} = 0$.

To show that

$$u_r\, h_o \equiv \text{hcf}\ (h_r,\ k_r) = h_r \qquad \text{if } k_{r+1} = 0$$
$$(u_r + v_r)h_o \equiv \text{hcf}\ (h_r,\ k_r) = k_r \qquad \text{if } h_r \;\; = 0$$

we again use induction. Suppose

$$v_r\, h_o \equiv k_r \qquad (\text{mod } D)$$

$$u_r\, h_o \equiv h_r \qquad (\text{mod } D)$$

These are clearly true for $r = 0$. To show they are true for $r + 1$:

$$v_{r+1}\, h_o - k_{r+1} = (v_r - u_r \left[k_r/h_r\right])\, h_o - (k_r - h_r\left[k_r/h_r\right])$$
$$= (v_r\, h_o - k_r) - (u_r\, h_o - h_r)\left[k_r/h_r\right]$$
$$\equiv 0 \qquad (\text{mod } D)$$

$$u_{r+1}\, h_o - h_{r+1} = (u_r - v_{r+1} \left[h_r/k_{r+1}\right])\, h_o - (h_r - k_{r+1} \left[h_r/k_{r+1}\right])$$

$$= (u_r\, h_o - h_r) - (v_{r+1}\, h_o - k_{r+1})\left[h_r/k_{r+1}\right]$$
$$\equiv 0 \qquad (\text{mod } D)$$

Accordingly if $k_r = 0$

$$u_r\, h_o \equiv h_r = \text{hcf}\ (h_o,\ k_o)$$

and if $h_r = 0$

$$(u_r + v_r)\, h_o \equiv k_r + h_r \equiv k_r = \text{hcf}\ (h_o,\ k_o).$$

CHAPTER 2

EXPERIMENTS IN INTEGER PROGRAMMING

Chapter 2 : Experiments in Integer Programming

The method of integer programming described by Gomory in (ref. 1), often referred to as the Method of Integer Forms, permits of many variations. The most common of these consist of two operations, optimisation and adding constraints. First of all the problem, which is formulated as a linear programming problem, is solved as if it were a linear programming problem, using the methods outlined in Part 1 of Chapter 1. If the solution obtained is integer, the integer programming problem is solved. If the solution obtained is not integer a constraint of the type described in Part 5(i) of Chapter 1 is added. This has the property that it does not render infeasible any feasible integer point but does make infeasible the current optimal (non-integer) point. After this the problem is re-optimised in linear programming fashion, and the process repeated until an integer solution is found.

The author's experimentation in the Method of Integer Forms started along these lines. It originally consisted of trying different criteria for choosing constraints of the type described in Part 5(i) and adding them to a tableau held in the form of floating point numbers. It was soon realised that there was more to integer programming than merely choosing good constraints. Parts 2 and 3 of this chapter describe these other problems and how they were dealt with, and Part 4 describes some of the different constraints tried and compares their performance.

Firstly however we digress slightly upon the purpose of linear programming within an integer programming method.

Part 1 : Linear programming as a subset of integer programming.

The optimum feasible solution of a linear programming problem is defined by the identity of the variables which are basic in that optimum feasible solution. However if we wish to know the values of the variables in that solution or prove that it is indeed optimal we have to transform the tableau of the problem by a series of pivot operations. Thus pivoting performs two functions; firstly it gives us the values of the variables in the solution, secondly it indicates whether the solution is optimal and if not enables us to choose another pivot which will carry us nearer the optimum.

There is a direct analogy with integer programming. To define the optimum feasible solution of an integer programming problem we need to know what constraints have been added as well as the identity of the basic variables. However, these will not give us the value of the variables in the solution or establish its optimality or feasibility. (Here we use the term feasible to mean that all variables are integer-valued as well as non-negative). To this end we use linear programming. Every time a new constraint is added a linear programming routine is used firstly to establish the values of the variables and secondly to determine whether they are integer or not. If they are not the tableau enables a constraint to be chosen which will carry the solution nearer the optimum.

We now present an example which illustrates another aspect of iterating. When constraints are derived in the manner of Part 5(i) of Chapter 1 they represent a lower limit on a non-negative combination of the variables which are non-basic at that time. In other words when a new constraint is added it will not assist the choice of any succeeding constraints until its slack variable has been made basic by a pivot operation.

The problem consists of two constraints :

$$4x + 2y \leq 5 \qquad 2.1.1.$$

$$2x + 4y \leq 7 \qquad 2.1.2.$$

These define a convex region which contains three integer points (0,0) (0,1) and (1,0). The feasible integer space is bounded by the two implicit constraints $x \geq 0$, $y \geq 0$, and a new constraint,

$$x + y \leq 1. \qquad 2.1.3.$$

To obtain 2.1.3 from 2.1.1 and 2.1.2 we first divide 2.1.1 and 2.1.2 by 2 and round down the right hand sides to the nearest integer in the manner of Part 5(ii) of Chapter 1. These constraints then become

$$2x + y \leq 2 \qquad 2.1.4$$

$$x + 2y \leq 3$$

If now we add these new constraints, divide their sum by 3, and round down the right hand side to an integer value we obtain 2.1.3.

We have derived 2.1.3 from 2.1.1 and 2.1.2 by a two stage process and we shall show that it cannot be done in a single stage. For example if we add 2.1.1 and 2.1.2 and divide the sum by six we obtain

$$x + y \leq 2. \qquad 2.1.6$$

Part 5(ii) of Chapter 1 showed that any constraint generated by the Method of Integer Forms could be obtained by taking a linear combination of the original inequalities and any additions to them and rounding all coefficients down to integer values. But before additional constraints can be used to generate any further constraints their slack variables must first be eliminated from the basis, i.e. a pivot operation must be performed.

We may write a linear combination of 2.1.1 and 2.1.2. as

$$\lambda\ (4x + 2y) + (\lambda + \mu)(2x + 4y) \leq \lambda\, 5 + (\lambda + \mu)\, 7 \qquad 2.1.7$$

where $\lambda \geq 0$ and $\lambda + \mu \geq 0$, but otherwise are not restricted. Simplifying, 2.1.7 becomes

$$(6\lambda + 2\mu)\ x + (6\lambda + 4\mu)\ y \leq 12\lambda + 7\mu$$

We now attempt to find a number, ν, such that

$$\left[\frac{6\lambda + 2\mu}{\nu}\right] x + \left[\frac{6\lambda + 4\mu}{\nu}\right] y \leq \left[\frac{12\lambda + 7\mu}{\nu}\right]$$

is identical to 2.1.3. We observe by comparing right hand sides that

$$2\nu > 12\lambda + 7\mu \qquad 2.1.8$$

In order that the coefficient of x might be at least 1 we have

$$\nu \leq 6\lambda + 2\mu$$

which to be consistent with 2.1.8 requires that $\mu < 0$.
On the other hand for the coefficient of y to be at least 1 we have the condition.

$$\nu \leq 6\lambda + 4\mu$$

which to be consistent with 2.1.8 requires $\mu > 0$.

As μ cannot be simultaneously < 0 and > 0 we have shown 2.1.3 cannot be obtained directly from 2.1.1 and 2.1.2. In particular we note that having found an optimal and feasible solution in the linear programming sense, one cannot expect to find the integer solution by adding all possible constraints.

Part 2 : The linear programming structure of the experimental programmes

The author started experimenting with integer programming using an Algol programme received from Dr. J.C. Wilkinson of Liverpool University. This used the Simplex Method of linear programming to find a linear programming optimal solution and then added a constraint of the type derived by Gomory and described in Part 5(i) of Chapter 1. The cycle of linear programming optimisation and adding a constraint was repeated until an integer solution was found.

It was not long before problems of accuracy were encountered, and the purpose of this part is to present the difficulties met and the methods used to overcome them.

The first problem arose because the programme worked in floating point. Every constraint added contained coefficients of the form $\left[a_{ij}\right] - a_{ij}$, and these usually implied a loss of accuracy. This loss of accuracy was evident because every iteration the value of the determinant d, calculated as the product of the pivot elements, was printed, and this value often lost any resemblance to an integer, although it was supposed to be one. This loss of accuracy often prevented quite small problems being solved, for example the 4 x 5 Problem no. 10. 1 in Appendix B.

The remedy adopted to tackle this problem was to rewrite the programme using integer arithmetic throughout, employing the algebra of Part 2 of Chapter 1. This introduced the restriction that the original equations must have integer coefficients and that any constraints introduced during solution must represent constraints in the original space with integer coefficients (see Part 3 of Chapter 1). However, it is usual to adopt such restrictions in integer programming as it means that all slack variables are integer valued.

The use of integer arithmetic gives rise to another problem, that of integer overflow. There are two ways of approaching this problem. One is to attempt to avoid it by keeping the elements of the tableau as small as possible; the other is to wait until it occurs or is about to occur and take action then. Eventually the latter approach was taken. Checks were made for overflow while pivoting and if overflow occurred the tableau was restored to its form before the pivot operation was started. However, this approach was not taken immediately because the Algol language contains no built-in facilities to inform the object-programme when overflow occurs. KDF9 Algol tests for overflow but

terminates the programme if it occurs. To test for overflow in Algol it would be necessary to perform every calculation twice: first in real arithmetic to check that the answer is in range, secondly in integer arithmetic to retain accuracy.

It was considered that there was a clear case for using User Code to carry out pivot operations. This raised difficulties of its own for the Algol interpreter normally used for developing programmes did not accept user code bodied procedures. Instead the compiler had to be used and this only afforded one compilation per day in place of three using the interpreter. As a result the implementation of checks for overflow was postponed and attention turned to a techniques designed to lessen the chances of overflow. This technique was a method of scaling equations during solution of a problem.

To a large extent the size of the coefficients in a tableau of the form 1.2.9 are proportional to the size of d, and efforts to reduce the size of the coefficients were directed towards reducing the size of d. One way to do this is to scale the original equations before starting to solve a problem, that is to eliminate any common factors in them. The reason for this is that d is the determinant of part of the original matrix, and removing a common factor from a row of the original matrix will also remove it from d provided that this row has been incorporated into the determinant. This will be so if the slack variable associated with the row has been made non-basic.

As an example consider the following problem:

$$\begin{aligned} \text{minimise} = & - 2x_1 - 3x_2 \\ \text{subject to} \quad & 2x_1 + 4x_2 \leq 6 \\ & 3x_1 + 3x_2 \leq 5 \end{aligned}$$

Writing x_3 and x_4 for the slack variables we write this in tableau form and perform one pivot.

	1	x_1	x_2			1	x_1	x_3
z	0	-2	-3		z	18/4	-2/4	3/4
x_3	6	2	4*	⟹	x_2	6/4	2/4	1/4
x_4	5	3	3		x_4	2/4	6/4	-3/4

2.2.1.

If we were to scale the first inequality by 2 the same pivot operation would become

	1	x_1	x_2			1	x_1	x_3'	
z	0	-2	-3		z	9/2	-1/2	3/2	2.2.2
x_3'	3	1	2*	⇒	x_2	3/2	1/2	1/2	
x_4	5	3	3		x_4	1/2	3/2	-3/2	

All the numerators save those in the last column are now half the size they were previously.

All this is fairly obvious, but what is not so obvious is that this scaling can be done automatically at times other than before starting the process of solution. The transformed tableau 2.2.1 has d = 4 and 4 possible constraints. They have coefficients:

(-2, -2, -1), (0, 0, -2), (-2, -2, -3), (0, 0, 0).

We note the second of these has zero constant term and if we append it to the transformed tableau 2.2.1 and perform a pivot we obtain

	1	x_1	x_3			1	x_1	s	
z	18/4	-2/4	3/4		z	9/2	-1/2	3/2	2.2.3
x_2	6/4	2/4	1/4	⇒	x_2	3/2	1/2	1/2	
x_4	2/4	6/4	-3/4		x_4	1/2	3/2	-3/2	
s	-0	-0	-2/4*		x_3	0	0	-2	

This is now the same as the transformed tableau 2.2.2 except that it has an extra row.

There was bound to be a constraint with zero constant term because 2.2.1 could produce four constraints. As it was equivalent to 2.2.2 the constant terms had to be the same and so any constraint could only have two values for the constant term: -2 and 0. This means there must be at least one constraint with zero constant term apart from the null constraint (0, 0, 0). For as there are four constraints but only two constant terms there must be at least two distinct constraints with the same constant term. If these are subtracted they generate a constraint with zero constant term. For example (-2, -2, -1) subtracted from (-2, -2, -3) will generate (0, 0, -2)

If it is possible to scale the original equations it is of course better to do it at the start rather than using the method just outlined. However the method has value as it is often possible to use it even when the original equations have no common factor. An example of this is given is figure 2.2.1. on page 55.

Another aspect of scaling is that besides reducing the size of the coefficients in the tableau it reduces the number of possible constraints. Noting that $x_3 = 2x_3'$ as can be seen from a comparison of 2.2.1 and 2.2.2

we can write the two non-trivial constraints of 2.2.1
as $2x_1 + x_3 \geq 2$ and $2x_1 + 3x_3 \geq 2$,
or $2x_1 + 2x_3' \geq 2$ and $2x_1 + 6x_3' \geq 2$.
Tableau 2.2.2 will only produce the first of these, and the first is clearly more restrictive than the second. This is also illustrated in figure 2.2.1.

It is believed that reducing the number of constraints in this way will increase the proportion of 'good' constraints and hence the likelihood of choosing one. Figure 2.2.1 compares some of the corresponding constraints in the tableaux before and after scaling. This comparison also suggests that a good choice of constraint is more likely to result from a scaled tableau than an unscaled one.

However the last constraint of figure 2.2.1 has a scaled version that contains a positive coefficient and thus excludes part of the space which no constraint of the form we are considering would exclude. This suggests that the benefits of scaling from the point of view of choosing constraints would be difficult to prove. In any case it presupposes that the choice of constraint is random.

Let us state the algebra of scaling more formally. If at some point during solution the constant terms are p_i/d and the p_i and d have a common factor, say g, and furthermore the tableau can generate d possible constraints, then we can scale the tableau. For any constraint can be derived by taking an integer linear combination of the rows of the tableau and deriving the remainders modulo d. Thus each constraint will have a constant term which is a multiple of g. As there are d constraints but only d/g constant terms it follows there are at least two distinct constraints with the same constant term. The difference of these constraints will generate a constraint with zero constant term. Adding this to the tableau and pivoting in the normal way will reduce the value of d without altering the values of the basic variables.

In practice such constraints were derived by searching the tableau for a constant term whose numerator p_i, had a factor, g, common with d, but whose associated row also contained a coefficient whose numerator a_{ij} did not contain the factor g. This row was then multiplied by the integer d/g and this generated the constraint.

To return to the original purpose of scaling. Scaling was introduced partly to assist the choosing of a cut (the constraints used in scaling are

Figure 2.2.1: An example of scaling

Minimize $10x_1 - 111x_2$

Subject to $-12x_1 + 109x_2 \leq 420$

$x_1 + x_2 \leq 20$

In tableau form:

	1	x_1	x_2		1	x_1	x_3
z	0	10	-111	z	46620/109	-242/109	111/109
x_3	420	-12	109*	x_2	420/109	-12/109	1/109
x_4	20	1	1	x_4	1760/109	121/109*	-1/109

	1	x_4	x_3
z	460	2	1
x_2	60/11	12/121	1/121
x_1	160/11	109/121	-1/121
s_1	0	-11/121	-11/121*

Constraint generated by taking 11 times the x_2 row.

This gives

	1	x_4	s_1
z	460	1	11
x_2	60/11	1/11	1/11
x_1	160/11	10/11	-1/11
x_3	0	1	-11

The extra constraint is equivalent to $-x_1 + 10x_2 \leq 40$ in the original tableau. In terms of the optimal tableau it can be written $-x_4 - x_3 + 11s_1 = 0$, i.e. $x_4 + x_3$ is a multiple of 11. One is in fact adding the two original constraints to get $-11x_1 + 110x_2 \leq 440$, and then dividing it through by 11.

The two tableaux do not in general generate the same constraints. For example,

$\frac{-5}{11} \geq \frac{-12}{121}x_4 \; \frac{-1}{121}x_3$ becomes $\frac{-5}{11} \geq \frac{-1}{11}x_4 \; \frac{-1}{11}s_1$

$\frac{-5}{11} \geq \frac{-23}{121}x_4 \; \frac{-12}{121}x_3$ becomes $\frac{-5}{11} \geq \frac{-1}{11}x_4 \; \frac{-12}{11}s_1$

$\frac{-6}{11} \geq \frac{-120}{121}x_4 \; \frac{-10}{121}x_3$ becomes $\frac{-6}{11} \geq \frac{-10}{11}x_4 \; \frac{-10}{11}s_1$

$\frac{-6}{11} \geq \frac{-109}{121}x_4 \; \frac{-120}{121}x_3$ becomes $\frac{-6}{11} \geq \frac{+1}{11}x_4 \; \frac{-120}{11}s_1$

not cuts in the strict sense of the word), and partly to try and avoid overflow. To both these ends scaling was carried out every time a new rational optimum solution was reached.

This enabled a larger type of problem to be tackled and solved. Inevitably problems arose which were abandoned because of overflow, and eventually a machine code subroutine was written to perform the pivot operation and test for overflow, and if necessary reconstruct the matrix. Rather than add a cut at a non-optimal solution an attempt was made to scale the matrix, and only if this was unsuccessful was a cut generated. As will be seen from the tables in Appendix C even this procedure failed. Once (Problem 1: programme BH9) overflow occurred when pivoting on a cut, and twice (Problem 6: programmes BHE and BHF) overflow occurred when d was equal to one and no cut could be added. These were rare happenings and no attempt to get past the difficulty was made. The difficulty could have been overcome by searching for alternative pivots or introducing rows of the sort Gomory generates in his all-integer algorithm.

At the same time as the means of combating overflow were being developed the author was suspecting more and more that the programme was prone to looping or circling. The evidence for this was that a series of rational solutions had the same value of the cost function.

Looping becomes possible when there are zero coefficients in the cost function. For if a pivot is chosen from a column with a cost coefficient of zero, the cost function will not change. If a succession of such pivots returns the tableau to a previous state it will continue to do so ad infinitum. There are two sorts of looping; one can happen in linear programming and the other in integer programming.

Looping in linear programming is moving from one infeasible or non-optimal basis to another and never reaching a feasible or optimal one. Examples of this have been constructed by A.J. Hoffman and E.M.L. Beale (ref. 4, pp 229-230).

Looping in integer programming is moving from one feasible and optimal solution to another and never reaching an integer one. At each rational solution a cut is added. After one pivot the slack variable associated with it will be made non-basic, but if after two or more pivots, or after further cuts, the slack variable re-enters the basis and has a positive value at the next rational solution, it will be

discarded as redundant. An example of this is given in figure 2.2.2.

Although there was no direct evidence of looping when the programmes were being developed, they often gave the appearance of being lost in a maze of figures. When they eventually got out it was more by luck than design!

It might have been possible to avoid the danger of looping by revising the rule of discarding previous cuts, but the systematic approach of lexicography was used instead. (see Chapter 1, Part 6). At first the simple form using the tableau of equation 1.6.2. was used. Later, when the cuts being generated became sensitive to the order in which the basic variables were held, the full lexicographic method based on the tableau of equation 1.6.3 was used.

Figure 2.2.2: An example of looping in integer programming

1.

	1	s_8	s_6	z	
c	0	0	0	1	
z	0	0	0	-1	
s_1	16/9	1/9	2/3	0	<-
s_2	2/9	-1/9	-2/3	0	
s_3	22	4	6	0	
s_4	2	-4	-6	0	
s_5	2	0	1	0	
y	1/2	0	-1/2	0	
s_7	2	1	0	0	
x	13/18	-1/9	-1/6	0	
s_1'	-7/9	*-1/9	-2/3	0	

2.

	1	s_1'	s_6	z
c	0	0	0	1
z	0	0	0	-1
s_1	1	1	0	0
s_2	1	-1	0	0
s_3	-6	36	-18*	0
s_4	30	-36	18	0
s_5	2	0	1	0
y	1/2	0	-1/2	0
s_7	-5	9	-6	0
x	3/2	-1	1/2	0
s_8	7	-9	6	0

3.

	1	s_1'	s_3	z
c	0	0	0	1
z	0	0	0	-1
s_1	1	1	0	0
s_2	1	-1	0	0
s_6	1/3	-2	-1/18	0
s_4	24	0	1	0
s_5	5/3	2	1/18	0
y	2/3	-1	-1/36	0
s_7	-3	-3*	-1/3	0
x	4/3	0	1/36	0
s_8	5	3	1/3	0

4.

	1	s_1'	s_3	z
c	0	0	0	1
z	0	0	0	-1
s_1	0	1/3	-1/9	0
s_2	2	-1/3	1/9	0
s_6	7/3	-2/3	1/6	0
s_4	24	0	1	0
s_5	-1/3	2/3	-1/6*	0
y	5/3	-1/3	1/12	0
s_1'	1	-1/3	1/9	0
x	4/3	0	1/36	0
s_8	2	1	0	0

5.

	1	s_7	s_5	z	
c	0	0	0	1	
z	0	0	0	-1	
s_1	2/9	-1/9	-2/3	0	
s_2	16/9	1/9	2/3	0	<-
s_6	2	0	1	0	
s_4	22	4	6	0	
s_3	2	-4	-6	0	
y	3/2	0	1/2	0	
s_1'	7/9	1/9	2/3	0	drop
x	23/18	1/9	1/6	0	
s_8	2	1	0	0	
s_2'	-7/9	-1/9*	-2/3	0	

6.

	1	s_2'	s_5	z
c	0	0	0	1
z	0	0	0	-1
s_1	1	-1	0	0
s_2	1	1	0	0
s_6	2	0	1	0
s_4	-6	36	-18*	0
s_3	30	-36	18	0
y	3/2	0	1/2	0
x	1/2	1	-1/2	0
s_8	-5	9	-6	0
s_7	7	-9	6	0

Figure 2.2.2 continued

7.

	1	s_2'	s_4	z
c	0	0	0	1
z	0	0	0	-1
s_1	1	-1	0	0
s_2	1	1	0	0
s_6	5/3	2	1/18	0
s_5	1/3	-2	-1/18	0
s_3	24	0	1	0
y	4/3	1	1/36	0
x	2/3	0	-1/36	0
s_8	-3	-3*	-1/3	0
s_7	5	3	1/3	0

8.

	1	s_8	s_4	z
c	0	0	0	1
z	0	0	0	-1
s_1	2	-1/3	1/9	0
s_2	0	1/3	-1/9	0
s_6	-1/3	2/3	-1/6*	0
s_5	7/3	-2/3	1/6	0
s_3	24	0	1	0
y	1/3	1/3	-1/12	0
x	2/3	0	-1/36	0
s_2'	1	-1/3	1/9	0
s_7	2	1	0	0

9.

	1	s_8	s_6	z	
c	0	0	0	1	
z	0	0	0	-1	
s_1	16/9	1/9	2/3	0	
s_2	2/9	-1/9	-2/3	0	
s_4	2	-4	-6	0	
s_5	2	0	1	0	
s_3	22	4	6	0	
y	1/2	0	-1/2	0	
x	13/18	-1/9	-1/6	0	
s_2'	7/9	1/9	2/3	0	**drop**
s_7	2	1	0	0	

Rule for choosing a cut: take fractional parts from the row with largest "right hand side".

Rule for choosing a pivot row: most negative "right hand side"

Rule for choosing a pivot column: in the event of a tie: first column

Rule for discarding a cut: when it ceases to be binding at an optimal (not necessarily integer) solution

The problem will cycle interminably since tableau 9 differs from tableau 1 only in the order of the rows, and the order of the rows is immaterial in this particular example.

Figure 2.2.2. continued

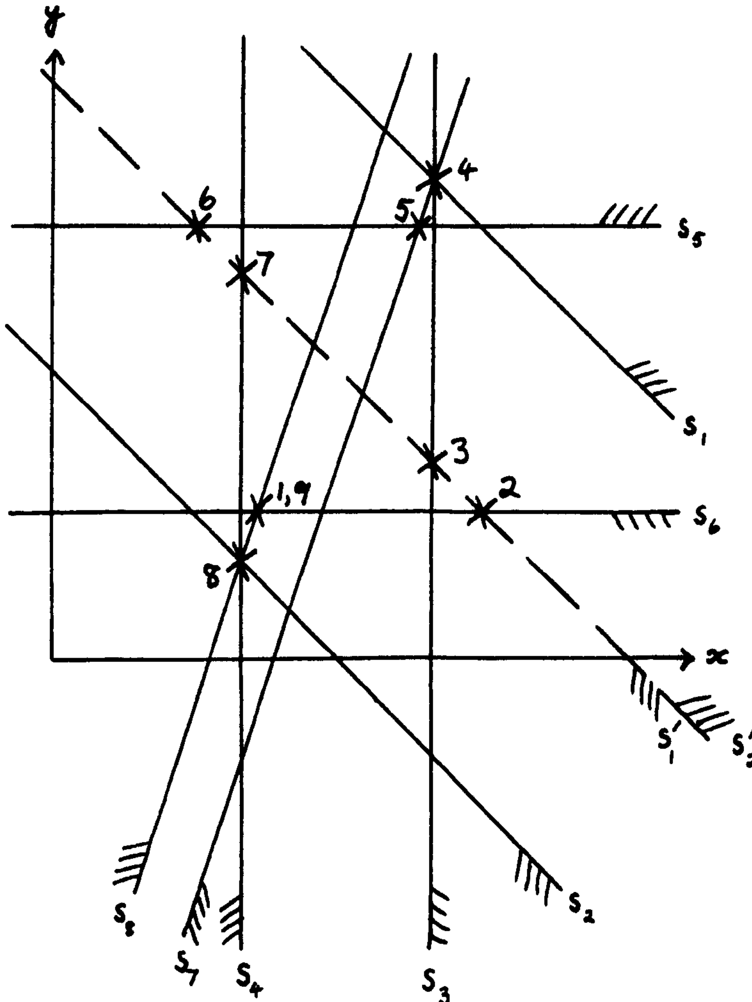
The tableaux represent an attempt to solve the problem:

Minimise z

Subject to

	i.e.	
$z \geq 0$		$z - z = 0$
$x + y \leq 3$		$x + y + s_1 = 3$
$x + y \geq 1$		$-x - y + s_2 = -1$
$36x \leq 48$		$36x + s_3 = 48$
$36x \geq 24$		$-36x + s_4 = -24$
$2y \leq 3$		$2y + s_5 = 3$
$2y \geq 1$		$-2y + s_6 = -1$
$9x - 3y \leq 7$		$9x - 3y + s_7 = 7$
$9x - 3y \geq 5$		$-9x + 3y + s_8 = -5$

In the following diagram the basic solutions encountered in the loop are marked with a cross and numbered.



Part 3 : Summary of the structure of the experimental programmes

If one took all the variations of programme described in Part 2 and multiplied them by a representative number of methods for choosing cuts one would end up with hundreds of programmes. In most of the programmes presented in Appendix D we have chosen to fix the methods used to find linear programming solutions and combat overflow in order to provide a valid comparison of different methods of choosing constraints. The two exceptions are programmes BGD and BH6 but even these differed from the rest only a little.

In consequence we present only one programme in full, and this is programme BHD. We now present a brief description of it.

The form of the data is specified at the start of Appendix D. After it is read in it is augmented by a negative unit matrix to enable every variable to appear as basic, and facilitate the use of the lexicographic method described in Part 6 of Chapter 1. This matrix is placed above the constraints contained in the data unless the data specifies otherwise.

Two linear programming procedures are used to find the optimum to the problem in rational numbers. The first is Intsimp which performs the Simplex Method, eliminating any artificial variables and optimising the tableau in such a way as to obtain the lexicographic optimum. This is followed by Dintsimp which performs the Dual Simplex Method, iterating until the constant terms are non-negative while maintaining lexicographic optimality. As mentioned in Part 1 of Chapter 1 these procedures enable the introduction of artificial variables to be avoided except when equalities are present.

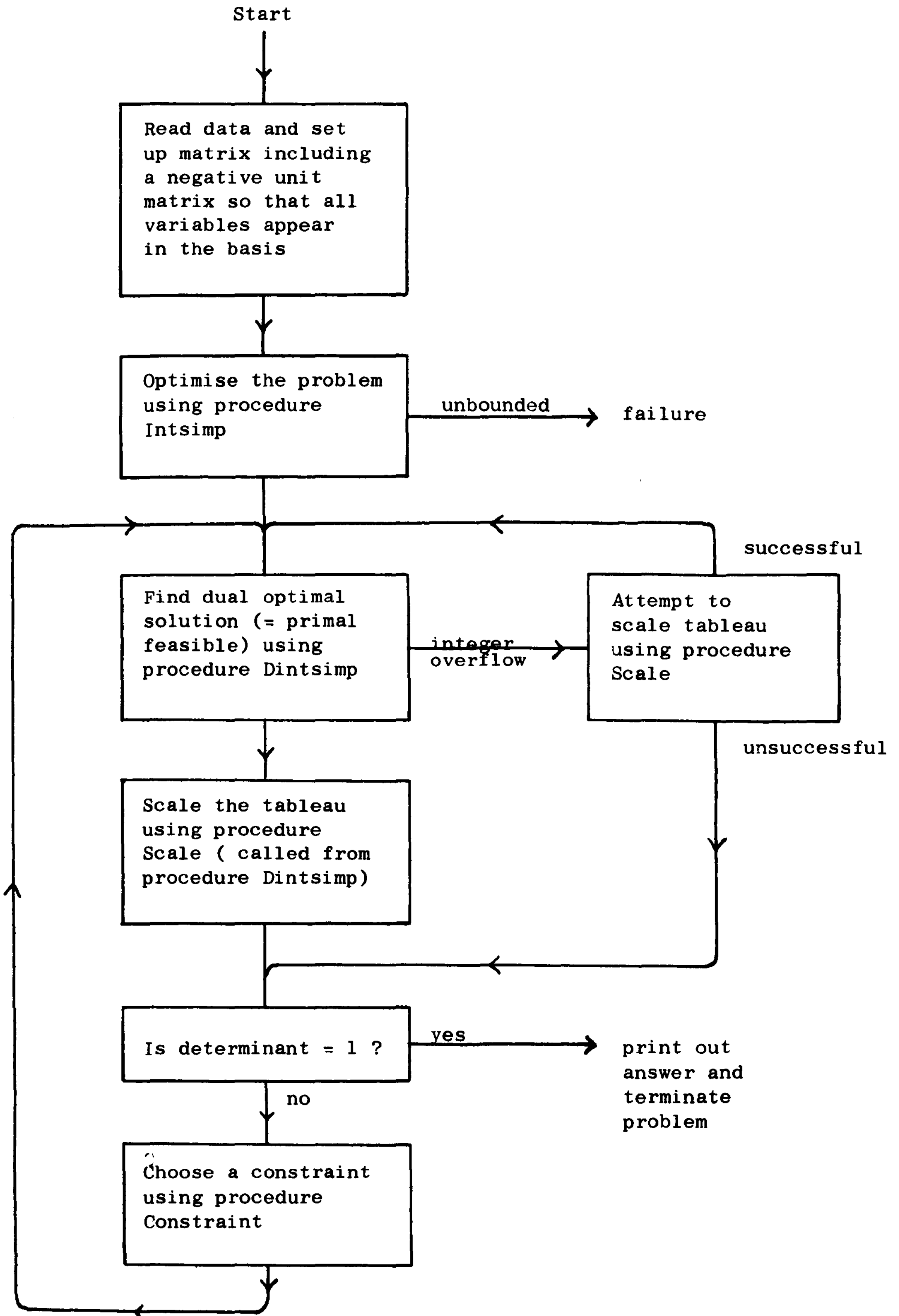
Once a feasible and optimal solution in rationals is found procedure Scale is used to scale the equations as described in Part 2 of this Chapter by adding constraints with zero constant terms, if any can be found. If after this the determinant, d, is equal to 1 the problem is solved.

At this point procedure Constraint is used to choose a constraint and add it to the tableau. Iteration then starts by returning to procedure Dintsimp to obtain a new feasible and optimal solution.

During pivoting a test is made for overflow. If it occurs the tableau is restored to its form before attempting to pivot. Procedure Scale is entered to try and reduce the value of d. If it is successful another attempt is made at pivoting. If scaling is unsuccessful, or overflow occurs again the tableau is treated as a scaled feasible optimal solution, i.e. procedure Constraint is used to generate another constraint. Following this another attempt is made to reach a feasible optimal solution.

Figure 2,3,1 presents a simplified flowchart of programme BHD.

Figure 2.3.1. An outline flowchart of programme BHD.



Part 4 : Description and comparison of the experimental programmes

Appendix C compares the performance of the programmes in appendix D when solving the problems contained in appendix B. The purpose of this part is to describe these programmes and to comment on their relative performance.

With the exception of programmes BGD and BH6 the programmes differed only in one procedure; procedure constraint. For this reason only one programme is given in full, and it is followed by the versions of procedure constraint used in the other programmes. As for programme BH6, this differed also in integer procedure pivot and the differing version follows the procedures constraint. Programme BGD differs from BHD in one line only and this is given in a comment on page 188. With the exception of BGD and BH6 the main body of the programmes is as described in the previous part. This part is concerned mainly with the various versions of procedure constraint.

As programmes BGD and BH6 are special cases the other programmes are discussed first. While the reader is entitled to his opinion, the author considered that of these other programmes BHD was the most consistent as well as often being the most efficient, particularly in the larger problems. For this reason BHD is described first, and the other programmes compared with it.

At the end of this Part figure 2.4.1 gives examples of the various methods of choosing constraints.

(a) Programme BHD. A cut is generated from the first row which has a non-integer right hand side. This row could be the cost function. Suppose the right hand side is a_{io}/D, and that $\frac{a_{io}}{D} = \left[\frac{a_{io}}{D}\right] + \frac{f_{io}}{D}$, where $0 < f_{io} < D$. 2.4.1

This will directly yield a constraint of the form

$$\underline{f}_i' \; \underline{x} \geq f_{io} \qquad 2.4.2.$$

If there is an integer, μ , such that

$$f_{io} < \mu \, f_{io} < D \qquad 2.4.3$$

we can multiply 2.4.2 by μ and take fractional parts once again. Taking fractional parts the second time will not alter the right hand side, because of 2.4.3, but might alter parts of the left hand side.

So the new constraint is

$$\underline{f}_i^{*\prime} \; \underline{x} \geq \mu \; f_{io} \quad , \text{ where } \underline{f}_i^{*} \leq \mu \; \underline{f}_i \qquad 2.4.4.$$

The cut 2.4.4 is at least as binding as 2.4.2.
The value of μ chosen is the largest possible.
In the notation of Algol:

$$\mu = (D - 1) \div f_{io} \qquad 2.4.5$$

A constraint of the form 2.4.2 has a special property. When the tableau is optimal and the constraint is taken from the cost function it has the effect of reducing the value of the cost function at least to the next integer below. To show this we denote the positive fractional part of an element of the cost function, a_{oj}/D, by f_{oj}/D:

$$\frac{a_{oj}}{D} = \left[\frac{a_{oj}}{D}\right] + \frac{f_{oj}}{D} \qquad 2.4.6$$

A constraint taken from this will have coefficients $-f_{oj}/D$. If we now pivot on the j th element of the constraint the cost function will change in value from a_{oo}/D to

$$\left[\frac{a_{oo}}{D}\right] + \frac{f_{oo}}{D} - \frac{f_{oo}}{D}\left(\left(\left[\frac{a_{oj}}{D}\right] + \frac{f_{oj}}{D}\right) \Big/ \frac{f_{oj}}{D}\right) \qquad 2.4.7$$

which will be less or equal to $\left[a_{oo}/D\right]$ according as $\left[a_{oj}/D\right]$ is greater or equal to zero.

If the cost function is integer-valued the constraint will be taken from the first row that is not integer valued. If we denote this as the i th row and the pivot column as the j th column, as before, we can use the same argument. If $\left[a_{ij}/D\right]$ is greater or equal to zero a_{io}/D will be reduced at least to the next integer below. If however $\left[a_{ij}/D\right]$ is negative it follows from our use of the lexicographic method that there will be a coefficient in the j th column in some row before the i th which is positive. The constant term of this row will then decrease in value. If it does not decrease as far as the next integer value below it only needs one more constraint to bring this about.

We thus have outlined an algorithm for making a systematic search for an integer solution. The argument is taken further in Chapter 3. The constraints added by programme BHD were of the form 2.4.4 where μ is as defined in 2.4.5. As 2.4.4 is at least as binding as 2.4.2 it has similar properties, the only difference being that a_{io}/D can be reduced below $\left[a_{io}/D\right]$ even when $\left[a_{ij}/D\right]$ is zero. Gomory suggests

this method in (ref 1, p 290).

(b) Programme BHM. This is similar to BHD in that the constraint is generated from the first row with a non-integer right hand side, but differs in that no multiple of the row is generated, the constraint is taken as it stands. This is the method for which Gomory constructed a finiteness proof in his paper (ref 1, p. 287).

In spite of the fact that at any given stage BHD will produce a cut at least as good as BHM, there was one problem (problem F5) in which BHM introduced fewer cuts than BHD. In two others (problems 7 and E4) BHM used more cuts but needed fewer pivots. While it is expected that the better the cut the more pivots are needed to reoptimise it would be surprising if experiments were to advocate deliberately choosing weak constraints. The examples show up one avoidable weakness of BHD, namely that if, in the notation of 2.4.4, $\underline{f}_i^* = \mu \underline{f}_i$, then the constraint will have a common factor and one or more extra constraints and pivots may be needed to eliminate it. However, in the majority of cases BHD took fewer pivots, fewer cuts, and less time.

(c) Programme BH9. This was a variation of BHD, the difference being that the search for a row with non-integer right hand side started with the first basic variable instead of the cost function. The object was to try and avoid zeros creeping into the cost function. The programme ended prematurely with two sets of data (problems 1 and 7) when overflow occurred immediately after adding a cut.

On the remaining sets of data its performance was similar to that of BHD.

(d) Programme BHQ. This, like BHD, generated a constraint from the first row with a non-integer right hand side. The Euclidean Algorithm was used to generate a constraint with the maximum possible right hand side. If the original right hand side is denoted by a_{io}/D, then the generated one is

$$\frac{D-\text{hcf}(a_{io}, D)}{D}$$

When a_{io} and D are mutually prime the method obtains the unique constraint with right hand side $(D-1)/D$.

In three problems (problems 3,9 and B4) BHQ was marginally better than BHD. In most cases, however, BHQ performed noticeably worse.

(e) Programme BHN. This was attempt to imitate the algorithm of Martin (ref 10). It first of all selected the row whose right hand side had the largest fractional part. It derived a constraint from this row,

without taking any multiple of it, and calculated the pivot column. However instead of pivoting it used the Euclidean Algorithm to determine the correct multiple of this constraint to make the previously calculated pivot element a minimum, this being usually minus one. The constraint calculated from this multiple of the original constraint was added and reoptimisation performed in the same way as in the other programmes.

This was not a very good approximation to Martin's algorithm, the main point of which was that it did not use the lexicographic dual simplex method. Instead it used the freedom of choice of optimum solution when there are zeros in the cost function to try and find an optimum with a small value of the determinant, D.

It did this by pivoting on the element mentioned in the previous paragraph which was calculated to be as small as possible. This entailed use of a composite algorithm. It is discussed more fully in Chapter 4. Because BHN bore little resemblance to Martin's algorithm it was only tried on a small set of examples. BHN was superior to BHD on problems 10. 1 to 10. 4 and 8 and 9, but considerably inferior on the more exacting problems 6 and 7.

(f) Programme BHP. The idea behind this programme was to take some of the ideas in Martin's algorithm and modify them in the context of the overall lexicographic method. It was also, in a sense, an opposite of BHQ. Whereas BHQ generated a large value for the right hand side of the constraint, BHP choose the column which was lexicographically smallest and which was eligible for pivoting, and generated a constraint whose coefficient in this column was as small as possible.

In detail, BHP first located the first row in the tableau with a non-integer right hand side. It then made a note of the columns which had non-integer elements in this row. Any constraint generated from this row would have zero coefficients in the remaining columns. Of these selected columns, the one which was lexicographically smallest was chosen. (A particular column is lexicographically smaller than another column if, when comparing the elements of the two columns from the top downwards, the first element of the first column which differs from the corresponding element of the second column, is smaller than that element). The element which lay in the chosen column and row was then subjected to the Euclidean Algorithm to find a multiple of this row such that the constraint produced from it had as small an element as possible in this chosen column. This

constraint was added and the tableau reoptimised. The element on which all the attention had been placed was not necessarily pivoted on. The sort of lexicography being used had only one optimum and was independent of the individual pivots used in obtaining it.

In spite of their similarity BHN and BHP differed considerably in the examples. On the whole BHP was better than BHN. When BHP was compared with BHD it was not obvious that BHD was a superior programme. Of the 24 examples solved by both BHP and BHD, BHP had fewer pivots in 12 of them and BHD fewer pivots in 9. As for the number of cuts the situation was reversed; BHP had fewer in 8 problems, and BHD fewer in 12. Most of these examples only differed between BHP and BHD by a very few pivots. If our attention is restricted to those examples for which the number of pivots taken by BHP and BHD differed by 10% or more we find BHD had fewer pivots in 6 examples and BHP fewer in 5. If our attention is restricted to examples where one programme took more than twice as many pivots as the other there are only two, and in both of them BHD took fewer pivots. They are problem 2: BHD 45 pivots, 6 cuts; BHP 378 pivots, 75 cuts; and problem 7: BHD 129 pivots, 30 cuts; BHP 318 pivots, 113 cuts.

There is not really enough evidence to say BHD is preferable to BHP. All one can say is that there are indications that BHD is more consistent. If the pivots and cuts for all 24 examples are added up, thus giving greater weight to the bigger problems, we find BHD has a total of 3549 pivots and 290 cuts, and BHP a total of 4092 pivots and 471 cuts. It is interesting to note that BHD's ratio of pivots to cuts is 12.2:1, and BHP's 8.7:1.

(g) Programme BHE. This programme chose constraints by 'the crudest possible criterion', to quote Gomory (ref 1, p.292) that is it examined the right hand side of each equation in the tableau, and chose the one with the largest fractional part. The constraint was added without any modification.

In one problem (problem 6) the run had to be abandoned. Integer overflow occurred and as D was equal to one at the time the usual avoiding action of adding a cut was not possible. In two problems (10. 2 and F4) BHE took fewer pivots than BHD, but in 11 others it took more.

(h) Programme BHF. This bore the same resemblance to BHE as BHD did to BHM. For each row with a non-integer right hand side, a_{io}/D, we calculate the largest multiple, μ_i, of the fractional part, f_{io}, such that $\mu_i f_{io} < D$ (see equations 2.4.1 to 2.4.5). The row which has the largest value of $\mu_i f_{io}$ is multiplied by its μ_i and the constraint is taken from this multiple.

BHF took fewer pivots than BHD in 6 problems, and BHD fewer than BHF in 9. However BHF never took less than 33% fewer pivots than BHD, whereas in one case (problem 6) the run of BHF had to be abandoned (for the same reason as BHE) after taking four times as many pivots as BHD, and in another case (problem 7) BHF took 1086 pivots and 451 cuts as opposed to BHD's 129 pivots and 30 cuts. BHD would seem a better programme than BHF mainly on the grounds of consistency.

(i) Programme BH6. The purpose of this programme was to demonstrate the advantages of using a lexicographic system. The programme was the same as BHF except for <u>integer procedure</u> pivot, the procedure that chose the pivot. Normally if two columns had the same ratio of objective function coefficient to pivot row coefficient the first and if necessary subsequent constraint rows were used to break the tie. In BH6 the method of breaking the tie was simply to take the first column. There was one place in the programme where this rule was broken. In many linear programming suites the initial primal optimisation is performed in two phases; first the artificial cost function is optimised and secondly the proper cost function. A lexicographic method allows these two optimisations to be done in one phase. Accordingly the initial optimisation when optimising the artificial cost function broke any ties by reference to the proper cost function. In subsequent optimisations only the proper cost function was used when choosing pivots.

To the surprise of the author this programme actually went into an infinite loop in three of the problems. Looping in linear programming is regarded as something which is theoretically possible but which never happens. The explanation offered for this discrepancy is that BHF was a programme based on a rigorous lexicographic system and that BH6 was generated by relaxing just one part of this system. It still retained the part of the system whereby every variable in the initial tableau was placed on the right hand sides of the equations and their relative order never changed. It also retained the use of integer arithmetic. This in particular was intended to make computation exact and avoid such things as rounding error. In linear programming the use of floating point

and its associated inexactitude often means that two numbers which are supposed to be identical are not, and this presents an automatic method of breaking ties.

Looping starts when the tableau of numbers used in solving a problem is identical to a previous one. In BH6 the identity of variables on the right hand side was always the same, and the numbers in the tableau were always correct. It was much easier for the programme to repeat itself than if these other things had been allowed to vary. It would seem to suggest that no rigour is better than some!

(j) Programme BGD. This programme was based on BHD. When it has a cost function consisting entirely of ones, as with the 'covering theorem' problem (problem 6) it worked exactly the same as BHD. It was designed to exploit one of the advantages and avoid two of the disadvantages of BHD. The advantage was that BHD regards every row as a cost row which is used to break ties in the previous row, and the first disadvantage was the difficulty of ordering the rows in the tableau to avail oneself of this advantage. The second disadvantage was that its performance was considerably affected by the size of the cost function. It was found that when the cost function could be divided through by a common factor (compare problem 10. 1 with 10. 2 and problem 10. 3 with 10. 4) there was usually a saving in time.

When an analogous operation was performed on a cost function without a common factor there was a similar result. The cost function of problem 1 was divided through by 7.5 and each coefficient rounded to the nearest integer. (The number 7.5 was chosen to try and minimise the accuracy lost by rounding). BHD solved this modified problem with 693 pivots whereas it was still a long way from solving the original problem after 3546. More by luck than anything the optimum point of the modified problem was the same as that in the original problem.

BGD was an extension of this principle. It started by setting up the tableau used by BHD. It then preceded the cost function by a row obtained by dividing the cost function by 2 and rounding to the nearest integer. This in turn was preceded by a row obtained by dividing the cost function by 4 and rounding. The process was continued using successive powers of 2 until the coefficients were all zero. The problem was solved using the row corresponding to the highest power of 2 as cost function and using successive rows as tie breakers. When a solution was found the value of the original cost function was printed out. As this

was not necessarily the optimum the original cost function was then constrained to be at least one better than the solution obtained. If another solution was found the same process was repeated, if the problem was then infeasible it was thereby established that the last solution found was the optimal.

The greatest success of BGD was that it solved the Markowitz and Mann problem (problem 1) in four minutes whereas BHD was still a long way from the solution after 30 minutes. Generally there was not much to choose between BGD and BHD. Using the number of pivots as criterion for choosing between BGD and BHD, BGD obtained the optimal solution first in 13 problems as opposed to BHD's 9, but by the time BGD had proved the solution optimal it was only ahead in 11, as opposed to BHD's 10. Using time as the criterion BGD obtained the solution before BHD in 11 problems as opposed to BHD's 12, but after proving optimality was only ahead in 8, whereas BHD was ahead in 15.

BGD took longer to perform a pivot operation because the tableau contained more rows.

Perhaps the main point of interest concerning BGD was that it produced a feasible solution fairly quickly. In only three problems (B4, A5, C5) did BHD produce a solution before BGD, and even then BGD took only 10% longer in time to produce a solution, which in those cases happened to be the optimum.

BGD, as well as BHD, is discussed in greater detail in Chapter 3 and a numerical illustration is given in Part 4 of that chapter.

(k) Story and Wagner (ref 11) used a form of Gomory's All-integer algorithm (ref 2) to solve a formulation of the 3-machine job-shop sequencing problem. They ran several sets of data of which problems A4 to F6 are some. Their results, which give only pivots, are listed for comparison.

There was a remarkable correlation between the number of pivots taken by Story and Wagner's programme and BHD. Out of 18 problems run, which took pivots varying in number from 22 to over 1000, in only three cases did one programme take more than twice the number of pivots the other did. In two cases the ratio was nearly twice. This makes problem A5 a strange exception to the pattern. BHD needed no cuts to solve it, only 35 pivots. It was a simple linear programming problem. However the all-integer algorithm took 613 pivots.

Out of 18 problems BHD took fewer pivots in 10 of them. However this is not a valid comparison as a pivot operation in the all-integer algorithm requires less arithmetic than in Gomory's other algorithm, as used in BHD.

(1) This last section is concerned not with a programme for solving integer programming problems but rather with a method for enabling an existing programme to obtain approximate solutions to a problem.

One of the ideas behind programme BGD was to be able to obtain approximate solutions to a problem comparatively quickly by simplifying the cost function. It was realised however that when the artificial cost function contained zeros that were not in the original cost function they might cease to be valid approximations to the original cost function. This was considered to be the case in Problems A4 to F6 which are described in the first half of Appendix B. Although the variables representing the slack time on machine III have coefficients of 1 in the cost function which vanish when divided by anything greater than 2 they are vital to the formulation of the problem. When they are omitted from an artificial cost function that function represents not the total idle time on machine III but merely the idle time of machine III before it starts its first job.

It was argued that in this case it would be more effective to scale the whole problem and not merely the cost function. To test this out the numbers used in problems A4 to F6 were scaled by dividing each by 3 and rounding to the nearest integer. This scaled problem was solved using BHD and in 15 cases out of 18 took fewer pivots than the original problem. The answers to the two problems never differed by more than 2. BGD obtained an answer before the scaled problem in 7 cases out of 12, but this first answer was out by as much as 5 in some cases.

Figure 2.4.1 : An illustration of the methods described in Part 4 of Chapter 2.

The problem : Maximise $4x + 13y$

subject to $3x + 7y \leq 11$

$2x + 9y \leq 12$

We denote the slack variables by u and v and write the problem in tableau form

1.

	1	x	y
z	0	-4	-13
u	11	3	7
v	12	2	9*

2.

	1	x	v
z	156/9	-10/9	13/9
u	15/9	13/9*	-7/9
y	12/9	2/9	1/9

3. rational solution

	1	u	v
z	242/13	10/13	11/13
x	15/13	9/13	-7/13
y	14/13	-2/13	3/13

The constraints generated by taking the fractional parts of these rows are:

(z)	-8/13	-10/13	-11/13
(x)	-2/13	-9/13	-6/13
(y)	-1/13	-11/13	-3/13

The whole group of thirteen constraints is as follows, the first one being generated directly from the row corresponding to y and the remainder being successive multiples of it.

(-1/13, -11/13, -3/13)
(-2/13, -9/13, -6/13)
(-3/13, -7/13, -9/13)
(-4/13, -5/13, -12/13)
(-5/13, -3/13, -2/13)
(-6/13, -1/13, -5/13)
(-7/13, -12/13, -8/13)
(-8/13, -10/13, -11/13)
(-9/13, -8/13, -1/13)
(-10/13, -6/13, -4/13)
(-11/13, -4/13, -7/13)
(-12/13, -2/13, -10/13)
(0 0 0)

Figure 2.4.1 : continued

(a) Programme BHD.

The first row with a non-integer constant term is the cost function. Taking fractional parts we obtain the row (-8/13, -10/13, -11/13). As 8 is greater than 13/2 we cannot improve the constraint and so we add it to the tableau and perform one iteration:

3. z = 18 8/13

	1	u	v
z	242/13	10/13	11/13
x	15/13	9/13	-7/13
y	14/13	-2/13	3/13
s	-8/13	-10/13	-11/13*

4. z = 18

	1	u	s
z	18	0	1
x	17/11	13/11	-7/11
y	10/11	-4/11	3/11
v	8/11	10/11	-13/11

Note that if the cost row had given us for example the row (-3/13, -7/13, -9/13) we would have improved it by taking the largest multiple, μ , of it such that $3\mu < 13$. This multiple is 4, so we would have multiplied through by 4 to get (-12/13, -28/13, -36/13). Taking fractional parts of the negative of this gives us the row (-12/13, -2/13, -10/13).

(b) Programme BHM.

This produces a constraint by taking factional parts from the first row with a non-zero constant term. No multiple of it is considered. In this case it produces the same constraint as in (a).

(c) Programme BH9.

We choose the first row after the cost function with a non-integer constant term. This is the row corresponding to x and it has fractional parts (-2/13, -9/13, -6/13). We improve it in the same way as BHD and multiply it by -6 and take fractional parts again to give (-12/13, -2/13, -10/13). If we add this to the tableau and pivot we have

3. z = 18 8/13

	1	u	v
z	242/13	10/13	11/13
x	15/13	9/13	-7/13
y	14/13	-2/13	3/13
s	-12/13	-2/13	-10/13*

4. z = 17 6/10

	1	u	s
z	176/10	6/10	11/10
x	18/10	8/10	-7/10
y	8/10	-2/10	3/10
v	12/10	2/10	-13/10

(d) Programme BHQ

Figure 2.4.1 : continued.

We pick the first available row which is in fact the cost function. Taking fractional parts we obtain (-8/13, -10/13, -11/13). Integer procedure euclidalg tells us that $5 \times 8 \equiv 1 \pmod{13}$ and so $(13 - 5) \times 8 = 8 \times 8 \equiv 12 \pmod{13}$.
Multiplying the row by -8 and taking fractional parts again we obtain (-12/13, -2/13, -10/13). This is the same constraint as produced in (c).

(e) Programme BHN.

We select the row whose constant term has the maximum fractional part and work out the constraint of fractional parts. This gives us constraint corresponding to the cost row which is (-8/13, -10/13, -11/13). If this constraint were added as in (b) the pivot column would have been the last. We derive the constraint with coefficient -1/13 in the last column. Integer procedure euclidalg tells us that $6 \times 11 \equiv 1 \pmod{13}$ and so we multiply the above constraint by -6 and take fractional parts again to obtain the row (-9/13, -8/13, -1/13). Incorporating it :

3. z = 18 8/13

	1	u	v
z	242/13	10/13	11/13
x	15/13	9/13	-7/13
y	14/13	-2/13	3/13
s	-9/13	-8/13*	-1/13

4. z = 17 6/8

	1	s	v
z	142/8	10/8	6/8
x	3/8	9/8	-5/8
y	10/8	-2/8	2/8
u	9/8	-13/8	1/8

(f) Programme BHP.

We choose the first available row and take fractional parts. Again this is the cost row and it gives us (-8/13, -10/13, -11/13). We choose the smallest column, that is the one corresponding to u, and minimise the coefficient in this row. Integer procedure euclidalg tells us that $4 \times 10 \equiv 1 \pmod{13}$ and so we generate a constraint from -4 times the above row. We obtain (-6/13, -1/13, -5/13). Incorporating it:

3. z = 18 8/13

	1	u	v
z	242/13	10/13	11/13
x	15/13	9/13	-7/13
y	14/13	-2/13	3/13
s	-6/13	-1/13	-5/13*

4. z = 17 3/5

	1	u	s
z	88/5	3/5	11/5
x	9/5	4/5	-7/5
y	4/5	-1/5	3/5
v	6/5	1/5	-13/5

Figure 2.4.1: continued.

(g) Programme BHE.

We choose the row whose constant term has the largest fractional part and we add the constraint formed by its fractional parts to the tableau. This produces the constraint (-8/13, -10/13, -11/13) from the cost function as in (a).

(h) Programme BHF.

This extracts the fractional part of each constant term and takes the maximum multiple of each such that the numerator remains less than the denominator. From the constant term of the cost function we get 1 x (8/13) = 8/13. From the row corresponding to x we obtain 6(2/13) = 12/13 and from the y row we obtain 12(1/13) = 12/13. We choose the row which generates the largest and in this case it is the x row as we found it before the y row. 6 times the x row generates (-12/13, -2/13, -10/13) which has already been iterated upon in (c).

CHAPTER 3

THE TWO MOST EFFECTIVE ALGORITHMS

Chapter 3: The two most effective algorithms.

In this chapter we consider programme BHD from the point of view of a programme which having found a feasible optimum solution in rationals, adds a constraint and reoptimises, the constraint being such that the first basic variable with a non-integer value is reduced at least to the next integer below. We are not concerned here with the mechanics of avoiding and dealing with integer overflow. However we are concerned with the use of the lexicographic method of choosing pivots, for it is this which determines from which row a constraint is taken.

We also consider programme BGD as an extension of BHD.

Part 1: The significance of a lexicographic method.

Part 4 of Chapter 2 included a brief description of programme BHD. As described there the programme derives a constraint from the cost function whenever this has a non-integer value, and the constraint is such that the value of the cost function is reduced at least to the next integer below. With the exception of BHM and BGD,the other programmes described in Chapter 2 generated constraints which bore no guarantee of doing this, and it was found that very often they did not do this, especially when D was large. The 'obvious' constraint gave a consistent and often superior result.

Although such a constraint has a good immediate effect on the problem it tends to make the problem more difficult by introducing zero coefficients into the cost function. This is evidenced by the performance of programme BHD when solving problem 1, listed in Appendix C. As the difference between the solution in rationals and the solution at the point the run was terminated was 42.4 the cost function can have had a non-integer value after at most 43 iterations. For such a value gives rise to a constraint reducing the value at least to the next integer below. However 426 cuts were added when trying to solve the problem which indicates that on average it took 10 iterations to shift the cost function from an integer value. As all the 426 cuts had non-zero constant terms the cost function must have contained zero coefficients.

The reason for this is suggested in part by the formula for the transformation of a coefficient of the cost function a_{oj}/D during a pivot operation incorporating a new constraint:

$$\frac{a_{oj}}{D} - \frac{f_{oj}}{D}\frac{a_{oh}}{f_{oh}} \qquad 3.1.1$$

where we denote the j th coefficient of the constraint by f_{oj}/D and the pivot column by h.

As a_{oh}, f_{oj}, and f_{oh} are all non-negative the expression 3.1.1 will tend to decrease. Of course this does not apply to other pivot operations.

Given that the coefficients of the cost function do decrease in value we would expect zeros to appear eventually. For the coefficients of a new constraint are defined by

$$\frac{\mu\, a_{oj}}{D} = \left[\frac{\mu\, a_{oj}}{D}\right] + \frac{f_{oj}}{D} \qquad 3.1.2$$

where μ is the largest integer such that

$$\mu\left(\frac{a_{oo}}{D} - \left[\frac{a_{oo}}{D}\right]\right) < D.$$

If $\mu\, a_{oj}/D$ is so small that the middle term of 3.1.2 vanishes we have $f_{oj} = \mu\, a_{oj}$. If similarly $f_{oh} = \mu\, a_{oh}$ the expression 3.1.1 will vanish. Furthermore if there are columns, j, such $f_{oj} = \mu\, a_{oj}$ the pivot column will be chosen from among them for only they minimise the ratio a_{oj}/f_{oj} by which the pivot column is chosen.

Any ties are resolved by reference to subsequent rows.

Once a cost row has zero coefficients in it, it will remain unchanged until a pivot row, possibly an added cut, is chosen which has no negative coefficients in the columns with zero cost. Sometimes a large number of pivot operations can be performed and several cuts added without changing the value of the cost function. When this happens it is rather like being lost in a maze and there is a potential danger of looping. As mentioned in Part 2 of Chapter 2 the lexicographic method of Part 6 in Chapter 1 was used to avoid this danger.

One way of regarding lexicographic ordering is that the method effectively turns the basic variables into secondary cost functions. If any row of the tableau contains coefficients which are the first non-zero elements in a column, these coefficients must be positive, by definition of lexicographic ordering. As any pivoting operation which did not alter the rows above the row in question would have to have a pivot in one of these columns, it would reduce the value of the basic variable associated with the row. Hence any basic variable in the tableau is maximised if we regard the basic variables above it as fixed.

Accordingly if a cost function, optimised and at an integer value, does not define uniquely the value of the variables, we have a subspace to search for an integer point, and for this we use the first basic variable in the tableau as a subsidiary cost function. If this still does not define the value of the variables the second variable becomes a cost function, and so on. As the value of no basic variable may be increased without reducing the value of a basic variable higher in the tableau, it follows that if the full lexicographic system is used no feasible solution is visited twice.

The purpose of these remarks is to demonstrate that in a lexicographic system every variable is a cost function. Thus, if the primary cost function is at an integer value and the first basic variable is at a non-integer value, our immediate object is to reduce the value of this variable as much as possible. We know we can reduce it at least to the next integer below, and so we do this. If it is already at an integer value, we inspect the next variable and act in a similar way.

Part 2: The dependence of the rate of convergence upon the ordering of the basic variables in the tableau

One of the biggest problems of integer programming is the unpredictability and irregularity of convergence. But although the method just outlined still suffered from these faults, the main cause of them was apparent.

Taking the next constraint from the first row whose basic variable had a non-integer value always reduced the value of that basic variable at least to the next integer below. However, it often went no further, and it is conceivable for a variable in successive steps to take every feasible integer value consistent with a fixed integer value of the cost function. The number of such integer values can be arbitrarily big. For example, in the Markowitz and Mann problem (ref. 13)(problem 1 in Appendix B) the values of the slack variables can vary from 0 to about 50 when the value of the cost function is in the region of the integer solution. If the equations were multiplied through by 10, the slack variables would have ten times as many feasible integer values. Accordingly if the slack variables were the first basic variables in the tableau, we would expect the value of the cost function to stay stationary for many iterations.

On the other hand, the 'proper' variables, that is the variables with non-zero coefficients in the cost function, can only take two values, zero or one, for any given value of the cost function. If these variables were listed as the first basic variables in the tableau, then at any iteration one would simply look for the first non-integer value, and force that variable to zero. One would expect such a constraint to have a much bigger effect on the problem than one derived from a slack variable, and that as a result fewer constraints would be required to break the value of the cost function away from a given integer value. In practice this was found to be the case.

In the Markowitz and Mann problem it is quite easy to deduce that one has a better chance of getting a good cut from a 'proper' variable than from a slack variable, and since if all the 'proper' variables are integer the slack variables are integer also, we need never take a cut from a slack variable. However it is more difficult to decide which among the'proper' variables should come first. As the coefficients in the constraints are randomly distributed one might

consider it best in this case to put first the variables with the largest coefficients in the cost function, but when this was tried it did not make a startling improvement to the speed of calculation.

While these procedures proved very useful for the Markowitz and Mann problem, it is very difficult to generalize them. In general, we wish to place the more 'significant' variables first. Unfortunately, while one can intuitively accept the concept of significance, it is extremely difficult to define, let alone construct an algorithm for. One is probably seeking variables which produce a large decrease in the value of the lexicographic cost vector per unit decrease of their own value. Slack variables, that is variables with zero coefficient in the cost function, are usually a bad choice. However, variables with large coefficients in the cost function will not necessarily be a good choice if their coefficients in the constraints are also large.

We have now described the reasoning which lay behind the development of programme BHD. We go on to discuss the considerations which gave rise to programme BGD.

Part 3: The problem of the dual function of the cost row

When the variables were ordered so that the 'proper' ones came first in the tableau there was a great improvement. The progress of the calculation became systematic and regular in the sense that the amount of calculation required to pass from one integer value of the cost function to the next increased slowly as the solution was approached.

In Part 1 of this chapter it was demonstrated that every variable was a cost function. The reverse is also true, that the cost function is an integer variable, though not restricted in sign. This means that instead of dividing a tableau into cost function and constraints, as is usual, each variable of the tableau performs both functions, and variables only receive differing treatment if some, but not all, are restricted in sign.

This enables us to manufacture variables which have no direct relation to the problem but are 'significant', using the word in the same sense as in Part 2 of this chapter. Such variables can be placed immediately after the cost function in the list of basic variables. Such new variables can sometimes be generated by dividing the coefficients in the cost function by a number greater than one and rounding to the nearest integer. Such variables will be integer valued and approximately proportional to the value of the cost function itself. So for a given value of the cost function, this new variable will be extremely restricted in value.

All the techniques discussed so far in Parts 1 and 2 of this chapter have been designed to reduce the work needed to search the sub-problem associated with any given integer value of the cost function, but they have no bearing on the fundamental weakness of the method so far described. This is that the cost function itself may have to pass through a large number of integer points before the solution is reached. In some aspects the cost function is similar to a slack variable in that it is not essentially an integer variable, but is only integer because it is an integer combination of integer variables. And like a slack variable, if its coefficients are multiplied by ten say, it will have ten times as many integer values to pass through to reach a solution.

The converse is true, that if we were able to divide the coefficients of the cost function through by a common factor we would speed up the

calculation, but generally this cannot be done without altering the problem and its solution.

The theory demands that one function should serve two purposes in the problem, to be both the cost and also the first variable from which cuts are taken. If this function should not be the original cost function but a function derived from it as already described, namely dividing the coefficients by a number greater than one and rounding, we will of course by liable to get a different answer. But under certain conditions the new cost function will have fewer integer values to pass through before reaching its optimum, and thus reach it more rapidly, and also the integer point at which it has its optimum value will furnish the original cost function with a value which is not far from its optimum. Once such an answer is found it is noted, and a new constraint added which constrains the value of the (original) cost function to be better than the one just found. The process continues until the problem becomes infeasible in which case the last solution found is known to be optimal.

As with 'significant' variables, it is difficult to construct an algorithm to determine whether such scaling of the cost function will reduce the number of integer points to be searched and at the same time produce a reasonable approximate answer. If the divisor of the coefficients is such that no new zeros are created in the cost function, the new function will be a genuine approximation to the cost function, even though the ratio between two coefficients could change by as much as a factor of three. (For example both 2.9 and 1 become 1 when divided by 2 and rounded to an integer).

However, if a coefficient becomes zero the associated variable has no influence on the cost. If the size of the coefficient in the cost function is a true indication of the importance or significance of the variable this does not matter, and the technique can be applied. This is the case with the first variable in the Markowitz and Mann problem (problem 1 in Appendix B). On the other hand the cost function in the job-shop scheduling problem (ref. 11) (problems A4 to F6 in Appendix B) contains unit coefficients for variables which represent the slack time on the third machine between successive jobs. These variables form an important part of the objective function and cannot be omitted without seriously affecting it.

If a problem is susceptible to a scaling of the cost function its speed of solution will depend very much on the choice of divisor. If the divisor is small, say two, the cost function will pass through about half the number of values, and reach a solution which is reasonably close to the optimal. But a factor of two is not usually a satisfactory saving when one is concerned with integer programming! It is often preferable to use a large divisor and obtain a good solution rapidly as long as it is not too distant from the optimal. Probably a divisor somewhere between the two extremes would be most satisfactory. An example of this is given in Appendix C under problem 1. Programme BHD failed to solve the problem even after 30 minutes. However when the cost function was divided through by 7.5 and each element rounded to an integer the same programme solved the altered problem in less than seven minutes. Because the divisor of 7.5 had been carefully chosen the two problems had the same solution.

Such speculation or experimentation is not necessary if we exploit our ability to have several cost rows. Firstly, we do not actually replace the original cost function by a new one, we keep it as a secondary cost function. This ensures that if at the optimum of the generated function the original one has two solutions, then the better one will automatically be chosen. If the divisor of the generated function is small the function will produce a solution close to the optimal, but will not produce it rapidly. Accordingly we can precede this function by a second cost function with a larger divisor. And the second cost function can be preceded by a third, and so on.

The technique actually used was to precede the cost function by a function generated by a divisor of 2, and precede this by one generated by a divisor of 2^2 , and so on using increasing powers of 2 until the function vanished. Each function was generated from the original row, not the row preceding it.

Programme BHD and its extension to become programme BGD were tested on several problems and the results of these tests are given in Appendix C and commented upon in Part 4 of Chapter 2.

Part 4: A numerical illustration of the use of artificial cost functions

We can illustrate the discussion of Part 3. Figure 3.4.1 solves a simple problem in three ways.

The first method is that of programme BHD. The original tableau consists of five rows; the cost function, the two constraints representing $x \geq 0$ and $y \geq 0$, and the two explicit constraints of the problem. The tableau is optimised by the Simplex Method. Before a row is pivoted on it is copied to the bottom of the tableau and afterwards this extra row is discarded.

Tableau 3 contains the optimal solution to the linear programming problem. The cost function a has an integer value so a constraint is taken from the first variable with a non-integer value, in this case x. The coefficients of the constraint are taken from the positive fractional parts of the row corresponding to x. This constraint is added to the bottom of the tableau and pivoted upon. Although the extra row at the bottom is then discarded the constraint itself is retained as the row corresponding to v in tableau 4.

Tableau 4 is optimal and feasible but still non-integer, and the process of adding a constraint and reoptimising is repeated twice more to give an optimal integer solution in tableau 6.

We note that after the initial optimisation three constraints and three reoptimisations were needed, the cost function a successively taking the values 10, $9\frac{1}{4}$, $8\frac{1}{2}$ and 8.

The second method involves adding an artificial cost function to the problem. The function added has been chosen on the basis that it must have smaller elements than the proper cost function but must also be a reasonable approximation to it. As in the first method the tableau is optimal after two pivots to give tableau 3. However this time we examine the artificial cost function when looking for a constraint. As its value is non-integer we derive a constraint from it by taking fractional parts and this reduces its value to an integer.

Tableau 4 has integer constant terms and is optimal and feasible; moreover the original cost function is also optimal. We have found an optimal integer solution with one constraint instead of three. However we perform one more iteration to produce an integer matrix.

Method 3 illustrates the working of programme BGD. It is an extension and mechanisation of the ideas of method 2. Instead of one artificial cost function we have several which are obtained by dividing the original one by 2,4 and 8, and rounding to an integer.

After one pivot the tableau, but not the original cost function, is optimal. The solution is also integer, but one constraint and one pivot element are needed to obtain the integer matrix of tableau 3.

We now have a feasible integer solution, and although this solution happens to give its optimal value we still have to prove its optimality. To do this we add a constraint to the bottom of the tableau which constrains the value of the cost function to be at least one better than the value we have just obtained. This constraint becomes a permanent addition to the tableau. As this constraint renders the tableau infeasible we copy it to the bottom of the tableau and pivot on it.

Tableau 4 is optimal and feasible again but non-integer. After adding one constraint and performing two pivots we obtain the final tableau, tableau 6. This tells us that the problem now has no solution as one row has a negative constant term but no negative elements on which to pivot. Therefore the optimal integer solution is the one obtained from tableau 3.

We note that method 3 produced the optimal integer solution more quickly than the other two methods, but lost this advantage in proving the optimality of the solution. In fact it took longer to prove optimality of the integer solution than it did to find it.

Figure 3.4.1: A comparison of three different ways of solving a simple problem.

The problem: ~~Minimise~~ Maximise $z = 4x + 3y$

Subject to $3x + y \le 6$

$x + 2y \le 4$

We represent the slack variables by u and v.

Method 1: The method of programme BHD.

1.

	1	x	y	
z	0	-4	-3	
x	0	-1	0	
y	0	0	-1	
u	6	3	1	<-
v	4	1	2	
u	6	3*	1	

2.

	1	u	y	
z	8	4/3	-5/3	
x	2	1/3	1/3	
y	0	0	-1	
u	0	-1	0	
v	2	-1/3	5/3	<-
v	2	-1/3	5/3*	

3. optimal, feasible, non-integer, $z = 10$

	1	u	v	
z	10	1	1	
x	8/5	2/5	-1/5	<-
y	6/5	-1/5	3/5	
u	0	-1	0	
v	0	0	-1	
s_1	-3/5	-2/5	-4/5*	

4. optimal, feasible, non-integer, $z = 9\frac{1}{4}$

	1	u	s_1	
z	37/4	2/4	5/4	<-
x	7/4	2/4	-1/4	
y	3/4	-2/4	3/4	
u	0	-1	0	
v	3/4	2/4	-5/4	
s_2	-3/4	-2/4*	-3/4	

5. optimal, feasible, non-integer, $z = 8\frac{1}{2}$

	1	s_2	s_1	
z	17/2	1	1/2	<-
x	1	1	-1	
y	3/2	-1	3/2	
u	3/2	-2	3/2	
v	0	1	-2	
s_3	-1/2	0	-1/2*	

6. optimal, feasible, integer matrix $z = 8$

	1	s_2	s_3
z	8	1	1
x	2	1	-2
y	0	-1	3
u	0	-2	3
v	2	1	-4

Figure 3.4.1 continued

Method 2 : Augmenting the problem with an artificial cost function $z_1 = x + y$ and using the method of programme BHD.

1.

	1	x	y
z_1	0	-1	-1
z_0	0	-4	-3
x	0	-1	0
y	0	0	-1
u	6	3	1 <-
v	4	1	2
u	6	3*	1

2.

		u	y
z_1	2	1/3	-2/3
z_0	8	4/3	-5/3
x	2	1/3	1/3
y	0	0	-1
u	0	-1	0
v	2	-1/3	5/3 <-
v	2	-1/3	5/3*

3. optimal, feasible,
non-integer, z_0 = 10.

	1	u	v
z_1	14/5	1/5	2/5 <-
z_0	10	1	1
x	8/5	2/5	-1/5
y	6/5	-1/5	3/5
u	0	-1	0
v	0	0	-1
s_1	-4/5	-1/5	-2/5*

4. optimal, feasible,
integer solution, z_0 = 8.

	1	u	s_1
z_1	2	0	1
z_0	8	1/2	5/2 <-
x	2	1/2	-1/2
y	0	-1/2	3/2
u	0	-1	0
v	2	1/2	-5/2
s_2	0	-1/2*	-1/2

5. optimal, feasible,
integer matrix, z_0 = 8

	1	s_2	s_1
z_1	2	0	1
z_0	8	1	2
x	2	1	-1
y	0	-1	2
u	0	-2	1
v	2	1	-3

Figure 3.4.1 continued

Method 3: Augmenting the problem with several artificial cost functions as in the method of programme BGD.

Added cost functions :

$$z_3 = x$$
$$z_2 = x + y$$
$$z_1 = 2x + 2y$$

1.

	1	x	y
z_3	0	-1	0
z_2	0	-1	-1
z_1	0	-2	-2
z_0	0	-4	-3
x	0	-1	0
y	0	0	-1
u	6	3	1
v	4	1	2
u	6	3*	1

2. optimal, feasible, integer solution, $z_0 = 8$

	1	u	y	
z_3	2	1/3	1/3	<-
z_2	2	1/3	-2/3	
z_1	4	2/3	-4/3	
z_0	8	4/3	-5/3	
x	2	1/3	1/3	
y	0	0	-1	
u	0	-1	0	
v	2	-1/3	5/3	
s_1	0	-1/3	-1/3*	

3. optimal, feasible, integer matrix, $z_0 = 8$

	1	u	s_1	
z_3	2	0	1	
z_2	2	1	-2	
z_1	4	2	-4	
z_0	8	3	-5	<-
x	2	0	1	
y	0	1	-3	
u	0	-1	0	
v	2	-2	5	
$\bar{z}_0$	-1	3	-5	
$\bar{z}_0$	-1	3	-5*	

4. optimal, feasible, non-integer, $z_0 = 9$

	1	u	$\bar{z}_0$	
z_3	9/5	3/5	1/5	<-
z_2	12/5	-1/5	-2/5	
z_1	24/5	-2/5	-4/5	
z_0	9	0	-1	
x	9/5	3/5	1/5	
y	3/5	-4/5	-3/5	
u	0	-1	0	
v	1	1	1	
$\bar{z}_0$	0	0	-1	
s_2	-4/5	-3/5	-1/5*	

5. optimal, infeasible,
   integer matrix, z_0 =13

	1	u	s_2	
z_3	1	0	1	
z_2	4	1	-2	
z_1	8	2	-4	
z_0	13	3	-5	
x	1	0	1	
y	3	1	-3	
u	0	-1	0	
v	-3	-2	5	<-
$\bar{z}_0$	4	3	-5	
v	-3	-2*	5	

6. infeasible, no solution.

	1	v	s_2	
z_3	1	0	1	
z_2	5/2	1/2	1/2	
z_1	5	1	1	
z_0	17/2	3/2	5/2	
x	1	0	1	
y	3/2	1/2	-1/2	
u	3/2	-1/2	5/2	
v	0	-1	0	
$\bar{z}_0$	-1/2	3/2	5/2	<-
$\bar{z}_0$	-1/2	3/2	5/2	

Part 5: Aspects of the algorithm which would benefit from further research

(a) The residual freedom of choice of constraint

The basic principle of the algorithm is to optimise the linear programming problem in such a way that each variable is maximised subject to the variables preceding it in the tableau remaining at their respective maxima. The first variable in the tableau, (the cost function being regarded as a variable), which is not at an integer value, is then reduced to at least the next integer below by a cut.

There are as a rule several cuts which will reduce the value of the variable at least to the next integer below. If in the terminology of equations 2.4.1. and 2.4.3., f_{io} is the fractional part of the value of the basic variable, there will be μ such cuts, where μ is the largest integer such that

$$\mu f_{io} < D. \qquad 3.5.1$$

In programmes BHD and BHM, μ was chosen to be as large as possible and to be equal to 1, respectively. Quicker results seemed to be obtained when μ was as large as possible, but there may be better values of μ . For example we might write $\mu = 2^{\lambda}$, where λ is as large an integer as possible consistent with 3.5.1. A constraint derived from this would have the property that it was at least as good as λ other constraints.

(b) Choosing the order of the variables

As explained in Part 2 of this Chapter, it is desirable to order the basic variables in the tableau according to their importance, or significance. While what is required is a quick and efficient way of selecting an ordering it would be a considerable advance simply to discover some property of the variables which affected it.

One possible way of deriving an order is based on the geometrical form of the problem. An optimal non-integer solution is at the vertex of a hypercone. The various variables in the problem will have values which lie on hyperplanes intersecting the hypercone. It is possible that the variables with the smallest range of feasible values will be the best ones to put at the head of the tableau.

In algebraic language we first solve the linear programming problem. We then constrain the cost function to be valued at the next integer below the rational optimum. For each variable in the problem we solve the subsidiary problems: maximise that variable and then minimise it. The variable with the smallest range of values is placed first in the tableau, and the remainder in order of their range of values.

If this method were successful it might be desirable to repeat it and rearrange the ordering at intervals during the calculation. For as the hyperplane defined by the cost function moves into the feasible space and cuts off more and more vertices, so will change the shape of the hypercone formed by the boundaries of the feasible region which intersect the hyperplane associated with the cost.

(c) The necessity of reducing the cost function to an integer value whenever possible.

The theory of programme BHD demands that the cost function should be reduced to an integer value whenever possible. For then if the first variable is not at an integer value the next cut will either reduce it at least to the next integer below or else reduce the cost function, in which case the next cut is chosen to reduce the cost function to at least the next integer below. If one does not reduce the cost function as often as possible the whole argument loses its validity.

Nonetheless when using the extended algorithm it does seem inefficient to have to use the wrong cost function in order that one might derive better cuts from it. One is tempted to try preceding the first artificial cost function by the original cost function in order that the original one is always optimal, but continuing to choose cuts starting at the artificial cost function.

(d) The manner of generating subsidiary cost functions

Programme BHD was extended into programme BGD by replacing the cost function, which we may denote by $\underline{c}'\,\underline{x}$, by a series of cost functions, which we may denote by $C\,\underline{x}$. The ith row of C is defined by

$$(C)_{i*} = \left\{ \frac{\underline{c}'}{\lambda_i} \right\}$$

where we use the curly brackets $\{\ \}$ to denote that each member of

$\underline{c}'/\lambda_i$ is rounded to the nearest integer. The λ_i are subject to the restriction

$$\lambda_{i-1} > \lambda_i > \lambda_{i+1} \qquad i = 2, \ldots, k - 1$$

where k is the number of rows of C, and

$$\lambda_k = 1, \lambda_1 \leq 2 \max (c_j).$$

The λ_i need not be integer.

Programme BGD defined λ_i by

$$\lambda_i = 2\lambda_{i+1}$$

A small modification of this would be

$$\lambda_{k-1} = 3, \lambda_i = 2\lambda_{i+1} - 1.$$

This will give us the series 1, 3, 5, 9, 17... instead of 1, 2, 4, 8, 16... The justification for this would be that when an integer is divided by an even number the maximum error when rounding is a half, but when it is divided by an odd number the error is always less than a half. In particular when dividing by 3 one gains a function with much smaller coefficients than when dividing by 2 but with no greater loss of accuracy.

More generally, there is scope for experiment in deciding the optimum number of λ_i and the distance between them. If the number of them is increased the cuts may become more efficient, but each pivot will take longer. It might be that a fibonacci series would be suitable, i.e.

$$\lambda_{k-1} = 2, \lambda_i = \lambda_{i+1} + \lambda_{i+2}, \qquad i = 1, \ldots k - 2.$$

This would of course increase the number of cost rows. Alternatively the number of cost rows could be reduced by use of the relation

$$\lambda_i = (k - i + 1)!$$

The last two sets of formulae give us series of 1, 2, 3, 5, 8, 13, 21,... and 1, 2, 6, 24, 120,... respectively.

(e) General computational procedure

The programmes described have been experimental. If they were going to be used on a routine basis for problem solving many changes would be necessary. Integer programming problems are often expressed in terms of large and sparse matrices, as are many linear programming problems. To deal with the latter first the inverse matrix method

(ref. 3, p.89) and then the product form of the inverse (ref. 4, p.200) were developed. These methods gain their efficiency of computation by evaluating as few elements of the transformed array as possible. In particular they use the Primal Simplex Algorithm and select pivot columns by reference only to the elements of the cost function, and pivot rows by reference to the constants column and the pivot column. The programmes in Appendix D use the lexicographic Dual Simplex Method.

Pivot rows are chosen by reference to the constants column, but pivot columns, besides referring to the cost function and pivot row, can require references to several other rows in order to break ties. It might not be economical to use the aforementioned methods for integer programming.

On the other hand the method of choosing additional constraints used in programmes BHD and BGD is ideal for use with the inverse matrix method.

The use of integer arithmetic would also cause problems. As pointed out in Part 7 of Chapter 1, the problems associated with integer arithmetic may well be fundamental to the problems of integer programming. Nevertheless the author considers that it was fortunate for him that KDF9 Algol permits the use of 39 bit integers, and KDF9 User Code permits the use of 96 bit integers. These word sizes were not always big enough for the problems tackled, and as computers seem to be standardising on 32-bit words, and compilers do not often provide facilities for multilength integers, this problem would merit further attention.

Associated with this problem is the question of how or whether to use the method of scaling the problem described in Part 2 of Chapter 2. It was used, firstly, after an optimum was reached, and secondly, after integer overflow occurred. With some of the methods of choosing constraints scaling may have improved the choice when applied at an optimum solution. This may also have been the case in programmes BHD and BGD. But as a method of counteracting overflow it may not have been so helpful.

CHAPTER 4

COMPARISON OF THE METHODS DESCRIBED IN THIS THESIS WITH THE WORK OF OTHER AUTHORS

Chapter 4: Comparison of the methods described in this thesis with the work of other authors

Part 1: Haldi and Isaacson (ref 9)

In their paper Haldi and Isaacson describe a method which differs very little from programme BHD. The author read their paper about the same time as he was forming his own ideas on the value of the algorithm. Although Haldi and Isaacson published their findings first an independent approach has enabled the author to view the problem from another angle. As Haldi and Isaacson acknowledge, Gomory was the first to describe the method (ref. 1, p.287), but he used it simply because it produced a neat finiteness proof. Some of the results of this thesis take the ideas of Haldi and Isaacson a little further.

Their method is actually that of programme BHM as described in Part 4(b) of Chapter 2. They do not suggest improving the constraints in the manner of programme BHD.

They make certain suggestions on problem formulation. They note that the wider the range of integer values the cost function has to pass through the longer the solution takes, and they "recommend that coefficients in the objective function be divided by multiples of 10 and rounded off whenever possible." (ref. 9, p.955). The burden of deciding whether it is possible or not lies on the user of the programme. There is no suggestion of an automatic procedure for doing this. They also recommend that columns and rows of the constraint matrix be rounded and scaled down wherever possible. This is to avoid numerical difficulties encountered in their use of floating point. In the integer arithmetic programmes contained in Appendix D these same problems would have caused the determinant, D, to have large values and possibly give rise to integer overflow. Reducing the size of the elements in the matrix would reduce the size of D. It is interesting to note that large numbers in the original matrix cause problems in both floating point and integer arithmetic solutions of the problem.

Haldi and Isaacson also realise that the ordering of the variables in the tableau is important. "Let the first variables in the data deck be those which, if their value should be changed by a unit amount, would cause the greatest net effect on the overall problem." (ref. 9, p.956).

Part 2: Martin (ref. 10)

The method of Martin is similar to the methods described in this thesis in that it is a direct extension of Gomory's algorithm (ref. 1). The steps of the method are:

(a) Optimise the linear programming problem.
(b) Choose a row with non-integer right hand side and derive the elementary constraint consisting simply of the fractional parts of the coefficients.
(c) Compute the column which would contain the pivot if this constraint were added to the tableau and the dual simplex algorithm were used to choose the next pivot.
(d) From this constraint generate a new one which has the smallest possible element in the above mentioned pivot column.
(e) Add this constraint to the tableau and pivot on this element.
(f) If the right hand sides are now integer, a sufficient condition for which is D = 1, return to step (a). Otherwise return to step (b).

The calculation (d) is carried out by a version of the Euclidean Algorithm. However instead of interating on D and the coefficient in the chosen pivot column to obtain the appropriate multiple of the row which would generate the desired constraint, the iteration is carried out on the whole row of coefficients. This is unnecessary and wasteful in time.

The more important ideas embodied in the method were taken and moulded into the format described in Part 2 of Chapter 2 to produce programme BHP. The steps of this programme correspond to that of Glenn Martin.

(a) Optimise. Programme BHP uses a lexicographic method.
(b) Choose a row with non-integer right hand side. If D is prime if does not matter which for each such row would generate every constraint. Programme BHP chooses the first row with a non-integer right hand side.
(c) Choose a column. It is questionable why one should make use of a constraint one is not going to apply. Instead programme BHP chooses the smallest column lexicographically speaking, providing of course the generating constraint has a non-zero element in that column.
(d) Generate the constraint with the smallest element in the chosen column. This produces the same result in both methods and has already been commented upon.

(e) Add this constraint to the tableau. As in a lexicographic system there is a unique optimum there is no point in forcing a pivot on any particular coefficient.

(f) Programme BHP always returns to (a) to reoptimise.

The main difference between the two methods is that where programme BHP uses a lexicographic method Martin's programme deliberately avoids it. In step (e) he chooses a pivot which is small, if not actually -1. This immediately ensures a matrix of small numbers. If the subsequent reoptimisation tends to keep the value of D small it might be of advantage in keeping numerical difficulties under control and in reducing the choice of constraints. However, no such explanation is offered in the paper. On the other hand there are definite advantages in using a lexicographic method.

The two methods were not compared computationally by the author. Step (e)in Glen Martin's algorithm would normally make the tableau non-optimal and infeasible, and the composite method needed to reoptimise the problem was not defined. However some aspects of the algorithm were incorporated into programme BHN. This was a modification of programme BHP such that the row chosen in step(b) was the one with largest fractional right hand side, and in step (c) the column was chosen by reference to the constraint derived in step (b). In the few problems solved by both BHP and BHN, BHP appeared to be superior. (See Part 4 of Chapter 2).

Part 3: Land and Doig (ref. 12)

The method of Land and Doig is a branch and bound algorithm which uses linear programming to calculate the bounds and provide information to help choose the next branch. The steps of the method are:

(a) Optimise the linear programming problem.

(b) Select an integer variable with a non-integer value.

(c) Branch on this variable. If we denote this variable by x_i and we have $x_i = a_{io}/D$, then the branches are

(i) add the constraints $x_i \geq [a_{io}/D] + 1$

(ii) add $x_i \leq [a_{io}/D]$.

(d) Put bounds on these two subproblems by solving them by linear programming. If either problem is infeasible, abandon it, otherwise augment it to the list of branches. If in (c), (i) is a better bound than (ii) it may be necessary at some stage to solve the subproblem with $x_i \geq [a_{io}/D] + 2$ as a constraint; if (ii) is better than (i) it may be necessary to solve with the constraint $x_i \leq [a_{io}/D] - 1$.

(e) Choose the subproblem with the best bound so far and return to (b).

Programme BHD has much in common with this method. We compare them step by step.

(a) Optimise. Programme BHD uses a lexicographic method.

(b) Select an integer variable with a non-integer value. Programme BHD always takes the first, if possible the cost function.

(c) Branch on this variable

(i) $x_i \geq [a_{io}/D] + 1$

(ii) $x_i \leq [a_{io}/D]$

If the previous branches, i.e. the variables higher in the tableau, are kept fixed (i) will be infeasible in programme BHD because x_i has been maximised subject to the higher variables. So programme BHD will always "branch" in one direction only.

(d) Bound the branch (es) by solving the linear programming sub-problem. Programme BHD will either obtain a solution

satisfying (ii) subject to the previous branches remaining fixed or, if no such solution exists, will automatically "branch" on one of the variables higher in the tableau, i.e. reduce it.

(e) Return to (b).

The advantage of Land and Doig's method over programme BHD is that it will cope with mixed integer problems. It is vital to the logic of programme BHD that a constraint is taken from the cost function whenever it is non-integer, but the cost function will not be constrained to be integer if it contains non-integer variables.

On the other hand if the method of Land and Doig used a lexicographic method of optimisation and selected variables for branching in the same order as programme BHD the two methods would follow similar courses. Whereas Land and Doig add simple constraints like $x_i \leq \left[a_{io}/D\right]$, programme BHD adds constraints which implicitly include the proviso that they only hold so long as the variables higher in the tableau do not change. This enables the programme to retrace its steps without having to store all previous partial solutions.

The close analogy between the two methods suggests that just as programme BHD was extended to become programme BGD by adding a set of artificial cost functions, so might the method of Land and Doig. The success of their method depends on finding variables on which to branch which will have a large effect on the cost function. In the pure integer case the cost function itself can be guaranteed to affect the cost function, but not in a large way. However scaled down versions of the cost function would, and it might well prove worthwhile to generate a set of cost functions in the same way as is described in Part 3 of Chapter 3.

The mixed integer problem is not so easy to generate integer variables for. Possibly the best that could be done would be to select that part of the cost function which consists of integer variables and derive a new set of integer variables from that.

Part 4: Backtrack methods

Programmes BHD and BGD can be interpreted as applying backtrack procedures. Such a description follows.

(a) Choose an integer variable. Calculate an upper bound for it by optimising the linear programming problem with this variable as cost function, and reduce it to an integer value if it is not already at one.

(b) Using linear programming, test whether the problem is still feasible. If not, proceed to (d).

(c) (i) If there are still some variables remaining in the tableau maximise the next variable and reduce it to an integer value if not already at one. Return to (b).

(c) (ii) If no variables remain a feasible integer solution has been found. Remember it if it is the best so far and proceed to (d).

(d) Backtrack: reduce previous variable by 1. If there is no previous variable the search is finished. Otherwise return to (b).

The programmes employ several short cuts. A constraint will sometimes reduce the value of a variable beyond the next integer below. Also when (b) finds a subproblem is infeasible it automatically reduces a previous variable in the tableau thus effecting a backtrack to that variable.

The above description is a fair description of programme BGD. To make it exact we have to specify how the first variables in the tableau are chosen. In addition programme BGD adds a sophistication whereby in (c)(ii) when a new feasible integer solution is found a constraint is added to ensure that any future feasible integer solution will be an improvement on the one just found.

If we define the variable chosen in (a) to be the cost function we obtain programme BHD. This has the property that the first feasible integer solution encountered is also the best.

There is little point in comparing these methods with other specific backtrack methods. The art of backtrack lies in inventing sophisticated short cuts which enable possible solutions to be enumerated implicitly rather than explicitly. Programmes BHD and BGD use linear programming for their short cuts.

An exposition of the principles of backtrack is contained in (ref. 14). It assumes that all variables are zero-one. It takes advantage of the fact that when adding variables to the list of those with assigned values they may be given any value initially rather than a predetermined one. This enables an algorithm to use heuristic techniques to get a reasonable solution quickly; it only becomes an exact algorithm after complete enumeration.

Programmes BHD and BGD are more concerned with getting an optimal solution. For this reason the first variable considered is either the cost function or an approximation to it.

Nevertheless it is possible that certain backtrack algorithms could be improved by introducing artificial cost functions as variables along the lines of programme BGD. This would depend very much on the individual algorithm.

REFERENCES

References.

1. Gomory, R.E. An Algorithm for Integer Solutions to Linear Programs. Recent Advances in Mathematical Programming, ed Graves, R.L., and Wolfe, P., McGraw Hill (1963), pp 269 - 302.
2. Gomory, R.E. An All-integer Integer Programming Algorithm. Industrial Scheduling, ed Muth, J.F., and Thompson, G.L., Prentice Hall (1963) pp 193 - 206.
3. Vajda, S. Mathematical Programming. Addison - Wesley (1961).
4. Dantzig, G.B. Linear Programming and Extensions. Princeton University Press (1963).
5. Mirsky, L. An Introduction to Linear Algebra. Oxford University Press (1955).
6. Wolfe, P. The Composite Simplex Algorithm. SIAM Review, vol 7, no 1, (Jan 1965) pp. 42 - 54.
7. Land, A.H. Talk given to the Mathematical Programming Study Group of the British Computer Society, 25th October, 1966.
8. Gomory, R.E. An Algorithm for the Mixed Integer Problem, The RAND Corporation, Paper P-1885, 22nd February, 1960.
9. Haldi, J., and Isaacson, L.M. A Computer Code for Integer Solutions to Linear Programs. Operations Research, vol 13, no 6, (Nov. 1965), pp 946-959.
10. Martin, G.T. An Accelerated Euclidean Algorithm for Integer Linear Programming. Recent Advances in Mathematical Programming, see [1], pp. 311-317.
11. Story, A.E., and Wagner, H.M. Computational Experience with Integer Programming for Job-shop Scheduling. Industrial Scheduling, see [2], pp. 207-219.
12. Land, A.H., and Doig, A.G. An Automatic Method of Solving Discrete Programming Problems, Econometrica, vol 28, no 3, (July 1960), pp. 497-520.
13. Markowitz, H.M., and Manne, A.S. On the Solution of Discrete Programming Problems. Econometrica, vol 25, (1957), pp. 84-110.
14. Geoffrion, A.M. Integer Programming by Implicit Enumeration and Balas' Method. SIAM Review, vol 9, no 2, (April 1967) pp. 178-190.
15. Obruca, A.K. Private communication.
16. Little, J.D.C., Murty, K.G., Sweeney, D.W., and Karel, C. An Algorithm for the Travelling Salesman Problem. Operations Research, vol 11, pp. 972-989 (1963).
17. Taussky, O. and Todd, J. Some Discrete Variable Computations -

Covering Theorem for Groups. Combinatorial Analysis, ed Bellman, R., and Hall, M., Proceedings of Symposia in Applied Mathematics, 10, American Mathematical Society, (1960) pp. 204-205.

18. Ferguson and Sargent, Linear Programming.

APPENDIX A

SYMBOLS NOTATIONS AND DEFINITIONS

Appendix A : Symbols, Notations and Definitions

Symbols and Notations

Symbol		Meaning
A,B,N	-	matrices
$\underline{x},\underline{y},\underline{b}$	-	column vectors
$\underline{x}',\underline{y}',\underline{c}'$	-	row vectors
$[B,N]$	-	partitioned matrix
B^{-1}	-	inverse of matrix B
$(B)_{ij}$	-	the element of B in row i and column j
$(B)_{i*}$, $(B)_{*j}$	-	the i th row and j th column of B
I_n	-	a unit matrix of dimension n. n is omitted where the dimension is apparent from the context.
O	-	a zero scalar, a zero vector, or a zero matrix, according to the context.
$\lvert B \rvert$	-	the determinant of B
d,D	-	the value of the determinant of the matrix which has implicitly been inverted at any stage of a linear programming problem. This matrix is usually denoted by B.
B*	-	the adjugate matrix of B, i.e. $(B^*)_{ij}$ is defined as the cofactor of $(B)_{ij}$ in B.
$\underline{e}_i$	-	a vector whose i th element is one and whose other elements are zero
$F \equiv G \pmod{h}$	-	every element of F - G is a multiple of h

$[a]$	- the largest integer not greater than a
$[a]_d$	- the largest multiple of d not greater than a
$[\underline{b}]$, $[\underline{b}]_d$	- each element of $\underline{b}$ is rounded down to an integer or a multiple of d
$n!$	- factorial n
$\underline{a} \geq 0$, $A \geq 0$	- every element of a and A is non-negative

DEFINITIONS.

Term	Definition
Simplex Method	- a method used to solve a linear programming problem when the constant terms are all non-negative.
Dual Simplex Method	- a method used to solve a linear programming problem when the coefficients of the cost function are all non-negative and all artificial variables have been eliminated from the problem.
Composite Method	- any method which caters for problems not catered for by the above.
Optimal	- having all non-negative coefficients in the cost function.
Feasible	- having all constant terms non-negative.
Dual feasible	- optimal
Dual optimal	- feasible
Tableau	- the matrix of numbers which is manipulated during solution of a problem.
Lexicographic	- ordered, taken in the order written.
Lexicographically positive	- applied to a vector this means that the first element of it which is non-zero is positive.
Lexicographically optimal	- applied to a tableau of a problem it means that every column is lexicographically positive.
Lexicographically greater than	- $\underline{a}$ is lexicographically greater than $\underline{b}$ means that $(\underline{a}-\underline{b})$ is lexicographically positive.

APPENDIX B

THE TEST DATA

PART 1

DESCRIPTION OF THE PROBLEMS

Appendix B: The test data.

Part 1: Description of the problems.

Problem 1: A production problem.

This problem was contained in a paper by Markowitz and Manne (ref. 13). It is a hypothetical production problem where a choice has to be made among 21 items to be manufactured subject to the limitations of six resources. Only one of each item can be chosen.

The coefficients of the constraints and cost row were chosen from a table of random numbers.

Problem 2: A two-dimensional knapsack problem.

This was contained in a paper by Weingartner and Ness. It is described as a two-dimensional knapsack problem and is of a similar form as problem 1 except that there are only two resources instead of six. However the coefficients are not random as can be seen from the high proportions of zeros in one constraint.

Problem 3, 4, and 5: Travelling salesman.

These problems are a formulation of the travelling salesman problem due to A.W. Tucker and described by Dantzig in (ref. 4, p. 547). We reproduce the derivation here.

Consider an n - city problem. Let $x_{ij} = 1$ if the salesman travels from city i to city j, and 0 otherwise. The problem is defined by the constraints.

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad (i = 1,\dots, n)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad (j = 1,\dots, n - 1)$$

$$\Sigma \; u_i \leq \tfrac{1}{2}\, n\, (n+1)$$

$$n x_{ij} + u_i - u_j \leq n - 1 \qquad (2 \leq i \neq j \leq n)$$

The first two sets of constraints define the assignment problem. The last set constrain any solution to be a tour provided all x_{ij} are 0 or 1. This is done by omitting city 1 from the equations and constraining the other links not to form a tour. If some of these links did form a tour, say of length k, we could sum the appropriate equations of the last set so that the u_i cancelled out, leaving

$$nk \leq k(n-1)$$

which is not possible. On the other hand any journey visiting cities 2 to n and not forming a tour will satisfy the last set of equations. To show this choose the values of u_i so the $u_i = t$ if city i is reached on the t-th step. Then the u_i will have the values 1,...n-1. If $x_{ij} = 0$ we have $u_i \leq u_j$ _ n-2 <n-1. If $x_{ij} = 1$ then $u_i = t$ and $u_j = t-1$ so that

$$nx_{ij} + u_i - u_j = n - 1.$$

The purpose of the third equation in the list was to provide an upper bound for the u_i which is demanded by the use of a lexicographic method.

Three problems were travelling salesman problem. Numbers 3 and 4 were 7 - city problems, the matrix of distances being symmetric and containing random numbers. The problems were obtained from A.K. Obruca (ref. 15). Number 5 was the non-symmetric 6 - city problem used as an example by Little, Murty Sweeney and Karel in their paper (ref. 16)

Problem 6: Covering theorem.

This problem is perhaps the most interesting of those presented here in that it is without doubt a genuine integer programming problem. A paper by Taussky and Todd (ref. 17) includes a description of the problem. The problem was tackled in collaboration with L.B. Wilson and J. Clowes of Newcastle University.

The specific problem posed in the paper was concerned with 5 entities, each of which could take on 3 values. The illustration given in the paper was of 5 football matches each having three possible results. To anticipate every possible outcome of the set of 5 matches one would have to be prepared for $3^5 = 243$ cases. However, if one is prepared to consider a subset of these 243 cases which is such that for any case there is a member of the subset which only differs from it in the outcome of one match one is left with much fewer possibilities. Such a subset is termed a covering and the problem was to find the smallest possible covering of these 5 matches.

To formulate the problem let us denote the three values of each match result by 1,2 and 3. We use the term element for each combination of five results and consider the 243 elements to be ordered as follows:

$$\begin{array}{ccccccccccl}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots\ldots\ldots\ldots\ldots \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & \\
1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 1 & \\
1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 & 1 &
\end{array}$$

Each element will cover itself and ten others. For example element (1,1,1,2,3) covers

$$\begin{array}{ccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 2 & 3 & 1 & 1 & 1 & 1 \\
1 & 2 & 2 & 2 & 3 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 1 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3
\end{array}$$

As $243/11 \doteq 22\ 1/11$ we deduce that a covering must consist of at least 23 elements. The question is what is the minimum size of a covering.

We associate a zero-one variable x_j with each element j such that x_j is one if j belongs to a given covering and zero otherwise. We define constants a_{ij} such that a_{ij} is one if elements i and j cover each other and zero otherwise. The problem can then be expressed as

$$\text{minimise } \sum x_j$$

$$\text{subject to } \sum a_{ij} x_j \geq 1 \qquad (i=1,\dots, 243)$$

$$\text{and } \quad x_j = 0 \text{ or } 1 \qquad (j=1,\dots, 243)$$

This problem was too large to be handled by any of the writer's programmes. Instead an auxiliary problem was solved which provided a lower bound for the number of elements in the covering. This auxiliary problem was one of a series that can be derived by making use of the special structure of the matrix (a_{ij}).

Let us write E_{243} for the matrix of coefficients (a_{ij}). It obeys the recurrence relation.

$$E_{3n} = \begin{bmatrix} E_n & I_n & I_n \\ I_n & E_n & I_n \\ I_n & I_n & E_n \end{bmatrix}, \quad E_1 = 1.$$

as may be verified by observing that in E_{243} each element covers and is covered by the elements 81 and 162 places after, the order of the elements being considered cydically, which gives rise to the I_n. Within in each of the three groups of 81 each element covers and is covered by those 27 and 54 places after. The reasoning is continued for the groups of size 27, 9, and 3, and E_1 is equal to 1 representing the fact that each element covers itself. As each element covers itself and 10 others we know that we have defined every non-zero element in the matrix.

The overall problem can be expressed as

$$\begin{bmatrix} E_{81} & I_{81} & I_{81} \\ I_{81} & E_{81} & I_{81} \\ I_{81} & I_{81} & E_{81} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{243} \end{bmatrix} \geq \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad \text{(243 elements)}$$

the partitioned matrix being the expansion of E_{243}. This can be weakened in two ways. Firstly we may add the three rows of the partitioned matrix and the corresponding rows of the vector on the right hand side to obtain.

$$\left[E_{81} + 2I_{81},\ E_{81} + 2I_{81},\ E_{81} + 2I_{81}\right] \begin{bmatrix} x_1 \\ \vdots \\ x_{243} \end{bmatrix} \geq \begin{bmatrix} 3 \\ \vdots \\ 3 \end{bmatrix} \quad \text{(81 elements)}$$

Secondly we may group the variables into 3s to obtain

$$\left[E_{81} + 2I_{81}\right] \begin{bmatrix} x_1 + x_{82} + x_{163} \\ \vdots \\ x_{81} + x_{162} + x_{243} \end{bmatrix} \geq \begin{bmatrix} 3 \\ \vdots \\ 3 \end{bmatrix} \quad \text{(81 elements)}$$

The value of $\sum x_i$ in the solution to this problem formed a lower bound to its value in the overall problems. The process of adding triples of rows and columns was repeated twice more to obtain the actual subproblem solved, which may be written algebraically as

$$\left[E_9 + 6I_9\right] \begin{bmatrix} x_1 + x_{10} + x_{19} + \dots + x_{235} \\ \vdots \\ x_9 + x_{18} + x_{27} + \dots + x_{243} \end{bmatrix} \geq \begin{bmatrix} 27 \\ \vdots \\ 27 \end{bmatrix} \quad \text{(9 elements)}$$

In Taussky and Todd's paper (ref. 17) it was stated that $\sum x_i$ must be at least 24 but need not be more than 27. A minimum solution to the sub-problem was found to be (5,2,2,2,3,3,2,3,3) giving a lower bound of 25 for $\sum x_i$, an increase of one on Taussky and Todd's figure.

The author also succeeded in solving the 27 x 27 subproblem, but this did not produce a better lower bound. The first attempts to solve the 27 x 27 subproblem were abortive because of trouble with integer overflow. The problem matrix has 5 s down the diagonal indicating a determinant of the order of 5^{27} or 5×10^{16}. However it was found possible to steer programme BHD round this stumbling block by adding redundant constraints containing small coefficients. An attempt was also made to solve the 81 x 81 problem by a method involving the use of the KDF9 linear programming package. This was unsuccessful, presumably because of the size of the determinant of the 81 x 81 problem which is of the order of 3^{81} or 10^{36}.

Problem 7: A problem with large coefficients.

This problem was given by Vajda as an example in (ref 3, p. 159). It has been scaled up to make all coefficients integer. Its main point of interest is that integer overflow occurred before a rational solution was found. Although the rational solution was never identified the integer solution was nevertheless found.

Problems 8 and 9 : Two very small problems.

Problem 8 was given by Vajda to illustrate integer programming (ref. 3, p. 199). It can be solved by adding a single cut.

Problem 9 was derived from problem 8 by slightly altering the ratios one to another of the coefficients in the first constraint. Its solution then required at least two cuts.

Problems 10.1, 10.2, 10.3 and 10.4: Four formulations of a product mix problem.

Problem 10.1 is due to Ferguson and Sargent (ref. 18). It concerns a hypothetical factory which can manufacture five different products. The products make varying demands on the labour resources in six different sections of the factory and the six constraints represent the limits of these resources. The problem is to select the product mix which will maximise the profit made by the factory.

The other three problems arose from the observation that the rows of problem 10.1 can be scaled down by factors varying from 5 to 60. Problem 10.2 is derived from problem 10.1 by scaling the cost function, problem 10.3 by scaling the constraints. In problem 10.4 all the rows have been scaled.

It is interesting to compare the numbers of pivots needed to solve the four forms of the problem.

Problems A4 to F6 : Job-shop scheduling.

These problems represent three-machine job-shop scheduling. n items are to be processed on each of three machines I, II and III, and in that order. The objective is to minimise the time elapsed between the start of the first item on machine I and the finish of the last item on machine III. Both the formulation and the data are taken from Story and Wagner (ref. 11).

The formulation takes advantage of the property of the three machine problem that there is an optimum solution in which the jobs are processed in the same order on each machine. Let us define variables x_{ij} to be such that x_{ij} is 1 if item i is scheduled in order - position j and 0 otherwise. The constraints start with the assignment problem matrix:

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad (j = 1,.., n)$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad (i = 1,.., n)$$

where n is the number of jobs to be processed. The formulation was altered slightly to avoid equality constraints:

$$\sum_{j=1}^{n} \sum_{i=1}^{n} x_{ij} \leq n$$

$$\sum_{i=1}^{n} x_{ij} \geq 1 \qquad (j = 1,.., n)$$

$$\sum_{j=1}^{n} x_{ij} \geq 1 \qquad (i = 1,.., n)$$

We must also have timing restrictions to ensure that each item is not processed by more than one machine at a time and that each machine is not processing more than one item at a time. We first define some extra variables.

Let s_{hk} = the slack time on machine h between the end of job k and the start of job k+1.

w_{hk} = the waiting time for job k between finishing processing on machine h and starting processing on machine h+1.

p_{hk} = the processing time for job k on machine h.

The timing constraints are derived by considering the time interval between the end of job k on machine h and the start of job k + 1 on machine h + 1. In this interval machine h must have processed job k + 1 so we can

express it as

$$s_{hk} + \sum p_{h,i}\, x_{i,k+1} + w_{h,k+1}.$$

At the same time machine h + 1 must have processed job k so we can also express the time interval as

$$w_{hk} + \sum p_{h+1,i} x_{ik} + s_{h+1,k}.$$

Equating these two we have

$$\sum p_{h+1,i} x_{ik} - \sum p_{hi}\, x_{i,k+1} - s_{hk} + s_{h+1,k} + w_{hk} - w_{h,k+1} = 0$$

where h attains values of 1 and 2 and k = 1,..., n - 1.

In the data submitted to the various programmes equals signs were avoided by replacing them by a greater than or equals sign in each constraint. To ensure that each constraint attained its lower bound the left hand sides of the constraints were summed and this sum was constrained to be less than or equal to zero.

The function to be minimised is the total idle time on machine 3:

$$\sum_{i=1}^{n} (p_{1i} + p_{2i})\, x_{i1} + \sum_{j=1}^{n-1} s_{3j}.$$

This has the same effect as minimising the total elapsed time.

The layout of data for a three job problem is given in the following table. The data for problems E4 and A5 are given in full; for the remaining problems only the values of the p_{ij} are given.

x_{11}	x_{21}	x_{31}	x_{12}	x_{22}	x_{32}	x_{13}	x_{23}	x_{33}	s_{21}	s_{22}	s_{31}	s_{32}	w_{12}	w_{13}	w_{22}	w_{23}		
q_1	q_2	q_3									1	1					=	z
1	1	1	1	1	1	1	1	1									$\leq$	4
1	1	1															$\geq$	1
			1	1	1												$\geq$	1
						1	1	1									$\geq$	1
1			1			1											$\geq$	1
	1			1			1										$\geq$	1
		1			1			1									$\geq$	1
p_{21}	p_{22}	p_{23}	$-p_{11}$	$-p_{12}$	$-p_{13}$				1				-1				$\geq$	0
			p_{21}	p_{22}	p_{23}	$-p_{11}$	$-p_{12}$	$-p_{13}$		1			1	-1			$\geq$	0
p_{31}	p_{32}	p_{33}	$-p_{21}$	$-p_{22}$	$-p_{23}$				-1		1				-1		$\geq$	0
			p_{31}	p_{32}	p_{33}	$-p_{21}$	$-p_{22}$	$-p_{23}$		-1		1			1	-1	$\geq$	0
r_{11}	r_{12}	r_{13}	r_{21}	r_{22}	r_{23}	r_{31}	r_{32}	r_{33}			1	1		-1		-1	$\leq$	0

where $q_j = p_{1j} + p_{2j}$ $r_{1j} = p_{2j} + p_{3j}$ $r_{2j} = p_{3j} - p_{1j}$ $r_{3j} = - p_{ij} - p_{2j}$

Variables s_{1j}, s_{i3}, w_{3j}, w_{i1} are omitted as they are either zero in an optimum solution or have no meaning in the problem.

The problem is to minimise z.

APPENDIX B

PART 2

THE PROBLEMS

Part 2 : The problems.

Note on the layout of the data.

Line 1 of ~~o~~ each problem contains a title enclosed between two '↑' symbols.

Line 2 contains the dimensions of the constraint matrix.

Line 3 contains the function to be minimised, the first number being an initial value for this function.

Lines 4 onwards contain the constraints of the problem. The first number of each of these lines is the constraint number. The symbol following the constraint number is $\leq$, =, or $\geq$ and determines the type of constraint. Then come the coefficients of the constraint followed by the constant term.

If the cost function or a constraint contained several zeros or ones in succession a shorthand was used whereby the number of zeros or ones appeared in the data followed by Z or U respectively.

For example,

```
↑ Problem 8↑
2 x 2;
0;   10; - 111;
1≤   -1;    10;   40
2≤    2U          20
```

represents : Minimise $10x - 111\, y$

subject to

$$-x + 10\, y \leq 40$$
$$x + y \leq 20$$
$$x, y \geq 0 \text{ and integer.}$$

```
↑Problem*1:**Markowitz*and*Mann*(ref*13)↑

27X21;

 0; -3;-47;-43;-73;-86;-36;-96;-47;-36;-61;-46;-98;-63;-71;-62;-33;-26;
        -16;-80;-45;-60;

 1≤    1;20Z 1;
 2≤  1Z1;19Z 1;
 3≤  2Z1;18Z 1;
 4≤  3Z1;17Z 1;
 5≤  4Z1;16Z 1;
 6≤  5Z1;15Z 1;
 7≤  6Z1;14Z 1;
 8≤  7Z1;13Z 1;
 9≤  8Z1;12Z 1;
10≤  9Z1;11Z 1;
11≤ 10Z1;10Z 1;
12≤ 11Z1;9Z 1;
13≤ 12Z1;8Z 1;
14≤ 13Z1;7Z 1;
15≤ 14Z1;6Z 1;
16≤ 15Z1;5Z 1;
17≤ 16Z1;4Z 1;
18≤ 17Z1;3Z 1;
19≤ 18Z1;2Z 1;
20≤ 19Z1;1Z 1;
21≤ 20Z1;    1;
22≤ 97;74;24;67;62;42;81;14;57;20;42;53;32;37;32;27; 7;36; 7;51;24; 400;
23≤ 16;76;62;27;66;56;50;26;71; 7;32;90;79;78;53;13;55;38;58;59;88; 400;
24≤ 12;56;85;99;26;96;96;68;27;31; 5; 3;72;93;15;57;12;10;14;21;88; 350;
25≤ 55;59;56;35;64;38;54;82;46;22;31;62;43; 9;90; 6;18;44;32;53;23; 320;
26≤ 16;22;77;94;39;49;54;43;54;82;17;37;93;23;78;87;35;20;96;43;84; 420;
27≤ 84;42;17;53;31;57;24;55; 6;88;77; 4;74;47;67;21;76;33;15;25;83; 400;
```

```
↑Problem*2:**2-dimensional*knapsack:**Weingartner*and*Ness*28*problem↑

30X28;

 0; -1898;-440;-22507;-270;-14148;-3100;-4650;-30800;-615;-4975;
       -1160;-4225;-510;-11880;-479;-440;-490;-330;-110;-560;
       -24355;-2885;-11748;-4550;-750;-3720;-1950;-10500;

 1≤ 45;0;85;150;65;95;30;0;170;0;40;25;20;0;
       0;25;0;0;25;0;165;0;85;0;0;0;0;100;          600;
 2≤ 30;20;125;5;80;25;35;73;12;15;15;40;5;10;
       10;12;10;9;0;20;60;40;50;36;49;40;19;150;    600;
 3≤    1U27Z 1;
 4≤ 1Z1U26Z 1;
 5≤ 2Z1U25Z 1;
 6≤ 3Z1U24Z 1;
 7≤ 4Z1U23Z 1;
 8≤ 5Z1U22Z 1;
 9≤ 6Z1U21Z 1;
10≤ 7Z1U20Z 1;
11≤ 8Z1U19Z 1;
12≤ 9Z1U18Z 1;
13≤ 10Z1U17Z 1;
14≤ 11Z1U16Z 1;
15≤ 12Z1U15Z 1;
16≤ 13Z1U14Z 1;
17≤ 14Z1U13Z 1;
18≤ 15Z1U12Z 1;
19≤ 16Z1U11Z 1;
20≤ 17Z1U10Z 1;
21≤ 18Z1U9Z 1;
22≤ 19Z1U8Z 1;
23≤ 20Z1U7Z 1;
24≤ 21Z1U6Z 1;
25≤ 22Z1U5Z 1;
26≤ 23Z1U4Z 1;
27≤ 24Z1U3Z 1;
28≤ 25Z1U2Z 1;
29≤ 26Z1U1Z 1;
30≤ 27Z1U    1;
```

```
↑Problem*3:**7-city*travelling*salesman↑

44×48;

 0; 37;63;80;49;84;30;37;40;60;20;50;19;63;40;19;21;70;
         59;80;60;19;40;85;78;49;20;21;40;55;39;84;50;
         70;85;55;55;30;19;59;78;39;55;0;0;0;0;0;0;

 1=    1;1;1;1;1;1;42Z 1;
 2=  6Z1;1;1;1;1;1;36Z 1;
 3= 12Z1;1;1;1;1;1;30Z 1;
 4= 18Z1;1;1;1;1;1;24Z 1;
 5= 24Z1;1;1;1;1;1;18Z 1;
 6= 30Z1;1;1;1;1;1;12Z 1;
 7= 36Z1;1;1;1;1;1; 6Z 1;
 8=     6Z1;5Z1;5Z1;5Z1;5Z1;5Z1;11Z 1;
 9=   1;12Z   1;5Z1;5Z1;5Z1;5Z1;10Z 1;
10= 1Z1;5Z1;12Z   1;5Z1;5Z1;5Z1; 9Z 1;
11= 2Z1;5Z1;5Z1;12Z   1;5Z1;5Z1; 8Z 1;
12= 3Z1;5Z1;5Z1;5Z1;12Z   1;5Z1; 7Z 1;
13= 4Z1;5Z1;5Z1;5Z1;5Z1;12Z   1; 6Z 1;
14≤ 42Z1;1;1;1;1;1; 28;
15≤  7Z7;34Z1;-1;4Z   6;
16≤  8Z7;33Z1;1Z-1;3Z 6;
17≤  9Z7;32Z1;2Z-1;2Z 6;
18≤ 10Z7;31Z1;3Z-1;1Z 6;
19≤ 11Z7;30Z1;4Z-1;   6;
20≤ 13Z7;28Z-1;1;4Z      6;
21≤ 14Z7;28Z   1;-1;3Z   6;
22≤ 15Z7;27Z   1;1Z-1;2Z 6;
23≤ 16Z7;26Z   1;2Z-1;1Z 6;
24≤ 17Z7;25Z   1;3Z-1;   6;
25≤ 19Z7;22Z-1;1Z1;3Z       6;
26≤ 20Z7;22Z  -1;1;3Z       6;
27≤ 21Z7;22Z      1;-1;2Z   6;
28≤ 22Z7;21Z      1;1Z-1;1Z 6;
29≤ 23Z7;20Z      1;2Z-1;   6;
30≤ 25Z7;16Z-1;2Z1;2Z    6;
31≤ 26Z7;16Z-1;1Z1;2Z    6;
32≤ 27Z7;16Z  -1;1;2Z    6;
33≤ 28Z7;16Z      1;-1;1Z 6;
34≤ 29Z7;15Z      1;1Z-1; 6;
35≤ 31Z7;10Z-1;3Z1;1Z  6;
36≤ 32Z7;10Z-1;2Z1;1Z  6;
37≤ 33Z7;10Z-1;1Z1;1Z  6;
38≤ 34Z7;10Z  -1;1;1Z  6;
39≤ 35Z7;10Z      1;-1; 6;
40≤ 37Z7; 4Z-1;4Z1; 6;
41≤ 38Z7; 4Z-1;3Z1; 6;
42≤ 39Z7; 4Z-1;2Z1; 6;
43≤ 40Z7; 4Z-1;1Z1; 6;
44≤ 41Z7; 4Z  -1;1; 6;
```

```
↑Problem*4:**7-city*travelling*salesman↑

44X48;*

 0; 42;78;51;63;71;70;42;47;65;87;75;85;78;47;68;93;62;
          81;51;65;68;25;23;20;63;87;93;25;38;20;71;75;
          62;23;38;20;70;85;81;20;20;20;0;0;0;0;0;0;

 1=     1;1;1;1;1;1;42Z 1;
 2=   6Z1;1;1;1;1;1;36Z 1;
 3=  12Z1;1;1;1;1;1;30Z 1;
 4=  18Z1;1;1;1;1;1;24Z 1;
 5=  24Z1;1;1;1;1;1;18Z 1;
 6=  30Z1;1;1;1;1;1;12Z 1;
 7=  36Z1;1;1;1;1;1; 6Z 1;
 8=      6Z1;5Z1;5Z1;5Z1;5Z1;5Z1;11Z 1;
 9=    1;12Z   1;5Z1;5Z1;5Z1;5Z1;10Z 1;
10=  1Z1;5Z1;12Z   1;5Z1;5Z1;5Z1; 9Z 1;
11=  2Z1;5Z1;5Z1;12Z   1;5Z1;5Z1; 8Z 1;
12=  3Z1;5Z1;5Z1;5Z1;12Z   1;5Z1; 7Z 1;
13=  4Z1;5Z1;5Z1;5Z1;5Z1;12Z   1; 6Z 1;
14≤  42Z1;1;1;1;1;1; 28;
15≤  7Z7;34Z1;-1;4Z   6;
16≤  8Z7;33Z1;1Z-1;3Z 6;
17≤  9Z7;32Z1;2Z-1;2Z 6;
18≤  10Z7;31Z1;3Z-1;1Z 6;
19≤  11Z7;30Z1;4Z-1;   6;
20≤  13Z7;28Z-1;1;4Z     6;
21≤  14Z7;28Z   1;-1;3Z  6;
22≤  15Z7;27Z   1;1Z-1;2Z 6;
23≤  16Z7;26Z   1;2Z-1;1Z 6;
24≤  17Z7;25Z   1;3Z-1;   6;
25≤  19Z7;22Z-1;1Z1;3Z      6;
26≤  20Z7;22Z  -1;1;3Z      6;
27≤  21Z7;22Z      1;-1;2Z  6;
28≤  22Z7;21Z      1;1Z-1;1Z 6;
29≤  23Z7;20Z      1;2Z-1;   6;
30≤  25Z7;16Z-1;2Z1;2Z     6;
31≤  26Z7;16Z-1;1Z1;2Z     6;
32≤  27Z7;16Z  -1;1;2Z     6;
33≤  28Z7;16Z      1;-1;1Z 6;
34≤  29Z7;15Z      1;1Z-1; 6;
35≤  31Z7;10Z-1;3Z1;1Z  6;
36≤  32Z7;10Z-1;2Z1;1Z  6;
37≤  33Z7;10Z-1;1Z1;1Z  6;
38≤  34Z7;10Z  -1;1;1Z  6;
39≤  35Z7;10Z      1;-1; 6;
40≤  37Z7; 4Z-1;4Z1; 6;
41≤  38Z7; 4Z-1;3Z1; 6;
42≤  39Z7; 4Z-1;2Z1; 6;
43≤  40Z7; 4Z-1;1Z1; 6;
44≤  41Z7; 4Z  -1;1; 6;
```

```
↑Problem*5:**6-city*travelling*salesman:**Little*et*al↑

32X35;

 0; 27;43;16;30;26;7;16;1;30;25;20;13;35;5;0;21;16;25;
        18;18;12;46;27;48;5;23;5;5;9;5;0;0;0;0;0;

 1=    1;1;1;1;1;30Z 1;
 2=  5Z1;1;1;1;1;25Z 1;
 3= 10Z1;1;1;1;1;20Z 1;
 4= 15Z1;1;1;1;1;15Z 1;
 5= 20Z1;1;1;1;1;10Z 1;
 6= 25Z1;1;1;1;1;5Z  1;
 7=   5Z1;4Z1;4Z1;4Z1;4Z1;9Z       1;
 8=  1;10Z    1;4Z1;4Z1;4Z1;8Z     1;
 9= 1Z1;4Z1;10Z    1;4Z1;4Z1;7Z    1;
10=  2Z1;4Z1;4Z1;10Z    1;4Z1;6Z   1;
11=   3Z1;4Z1;4Z1;4Z1;10Z    1;5Z  1;
12≤ 30Z1;1;1;1;1; 21;
13≤  6Z6;23Z1;-1;3Z    5;
14≤  7Z6;22Z1;1Z-1;2Z  5;
15≤  8Z6;21Z1;2Z-1;1Z  5;
16≤  9Z6;20Z1;3Z-1;    5;
17≤ 11Z6;18Z-1;1;3Z        5;
18≤ 12Z6;18Z    1;-1;2Z    5;
19≤ 13Z6;17Z    1;1Z-1;1Z  5;
20≤ 14Z6;16Z    1;2Z-1;    5;
21≤ 16Z6;13Z-1;1Z1;2Z      5;
22≤ 17Z6;13Z  -1;1;2Z      5;
23≤ 18Z6;13Z      1;-1;1Z  5;
24≤ 19Z6;12Z      1;1Z-1;  5;
25≤ 21Z6; 8Z-1;2Z1;1Z  5;
26≤ 22Z6; 8Z-1;1Z1;1Z  5;
27≤ 23Z6; 8Z  -1;1;1Z  5;
28≤ 24Z6; 8Z     1;-1; 5;
29≤ 26Z6; 3Z-1;3Z1; 5;
30≤ 27Z6; 3Z-1;2Z1; 5;
31≤ 28Z6; 3Z-1;1Z1; 5;
32≤ 29Z6; 3Z  -1;1; 5;
```

```
↑Problem*6:**covering*theorem↑

9X9;

0;  1;1;1; 1;1;1; 1;1;1;

1≥  7;1;1; 1;0;0; 1;0;0;  27;
2≥  1;7;1; 0;1;0; 0;1;0;  27;
3≥  1;1;7; 0;0;1; 0;0;1;  27;

4≥  1;0;0; 7;1;1; 1;0;0;  27;
5≥  0;1;0; 1;7;1; 0;1;0;  27;
6≥  0;0;1; 1;1;7; 0;0;1;  27;

7≥  1;0;0; 1;0;0; 7;1;1;  27;
8≥  0;1;0; 0;1;0; 1;7;1;  27;
9≥  0;0;1; 0;0;1; 1;1;7;  27;
```

```
↑Problem*7:**Vajda*(ref*3)*p.159↑

4X5;

0;     3;   7;   7;   5;   2;

1≥    83; 249;   4;  60;  51;    700;
2≥   246; 423; 793;  93;  26;   3000;
3≥    86;4050;  74; 308;2975;   4000;
4≥   201;  57;  16; 205; 400;   1200;
```

```
↑Problem*8:**Vajda*(ref*3)*p.199↑

2X2;

0;    10;-111;

1≤    -1;  10;   40;
2≤     1;   1;   20;
```

```
↑Problem*9:**perturbation*of*problem*8↑

2X2;

0;    10;-111;

1≤   -12; 109;  420;
2≤     1;   1;   20;
```

↑Problem*10.1:Ferguson*and*Sargent↑**

6x5;

0; -2000;-100-250;-400;-100;

1≤	800;	20;	20;	120;	30;	2000;
2≤	200;	10;	15;	30;	20;	1000;
3≤	300;	20;	40;	45;	10;	1000;
4≤	2400;	40;	240;	320;	160;	8000;
5≤	400;	30;	50;	80;	40;	2000;
6≤	900;	60;	240;	180;	120;	6000;

↑Problem*10.2:problem*10.1*with*cost*row*scaled↑**

6x5;

0; -40; -2; -5; -8; -2;

1≤	800;	20;	20;	120;	30;	2000;
2≤	200;	10;	15;	30;	20;	1000;
3≤	300;	20;	40;	45;	10;	1000;
4≤	2400;	40;	240;	320;	160;	8000;
5≤	400;	30;	50;	80;	40;	2000;
6≤	900;	60;	240;	180;	120;	6000;

↑Problem*10.3:problem*10.1*with*constraints*scaled↑**

6x5;

0; -2000;-100;-250;-400;-100;

1≤	80;	2;	2;	12;	3;	200;
2≤	40;	2;	3;	6;	4;	200;
3≤	60;	4;	8;	9;	2;	200;
4≤	60;	1;	6;	8;	4;	200;
5≤	40;	3;	5;	8;	4;	200;
6≤	15;	1;	4;	3;	2;	100;

↑Problem*10.4:problem*10.1*with*all*rows*scaled↑**

6x5;

0; -40; -2; -5; -8; -2;

1≤	80;	2;	2;	12;	3;	200;
2≤	40;	2;	3;	6;	4;	200;
3≤	60;	4;	8;	9;	2;	200;
4≤	60;	1;	6;	8;	4;	200;
5≤	40;	3;	5;	8;	4;	200;
6≤	15;	1;	4;	3;	2;	100;

↑Problem*E4:4-job*3-machine*job*shop*scheduling**

```
16X28;

 0; 10;13;14;20;15Z3U6Z

 1≤ 16U12Z 4;
 2≥    4U24Z 1;
 3≥  4Z4U20Z 1;
 4≥  8Z4U16Z 1;
 5≥ 12Z4U12Z 1;
 6≥   1;3Z1;3Z1;3Z1;15Z 1;
 7≥ 1Z1;3Z1;3Z1;3Z1;14Z 1;
 8≥ 2Z1;3Z1;3Z1;3Z1;13Z 1;
 9≥ 3Z1;3Z1;3Z1;3Z1;12Z 1;
10≥    5;13;7;10;-1;-1;-6;-9;8Z-1;2Z1;5Z-1;2Z  0;
11≥ 4Z5;13;7;10;-1;-1;-6;-9;5Z-1;2Z1;4Z1;-1;1Z  0;
12≥ 8Z5;13;7;10;-1;-1;-6;-9;2Z-1;2Z1;4Z1;-1;  0;
13≥    1;1;6;9;-9;-12;-8;-11;~~5Z1;4Z1;-1;4Z  0;~~ 8Z1;5Z-1;5Z    0;
14≥ 4Z1;1;6;9;-9;-12;-8;-11;5Z1;4Z1;-1;4Z  0;
15≥ 8Z1;1;6;9;-9;-12;-8;-11;2Z1;4Z1;-1;3Z  0;
16≤ 6;14;13;19;-4;1;-1;-1;-4;1;-1;-1;-10;-13;-14;-20;
        3Z3U2Z-1;2Z-1;  0;
```

↑Problem*A5:5-job*3-machine*job*shop*scheduling**

```
20X41;

 0; 13;36;34;7;13;24Z4U8Z

 1≤ 25U16Z 5;
 2≥    5U36Z 1;
 3≥  5Z5U31Z 1;
 4≥ 10Z5U26Z 1;
 5≥ 15Z5U21Z 1;
 6≥ 20Z5U16Z 1;
 7≥   1;4Z1;4Z1;4Z1;4Z1;20Z 1;
 8≥ 1Z1;4Z1;4Z1;4Z1;4Z1;19Z 1;
 9≥ 2Z1;4Z1;4Z1;4Z1;4Z1;18Z 1;
10≥ 3Z1;4Z1;4Z1;4Z1;4Z1;17Z 1;
11≥ 4Z1;4Z1;4Z1;4Z1;4Z1;16Z 1;
12≥     20;6;5;3;4;-8;-30;-4;-5;-10;15Z-1;3Z1;7Z-1;3Z  0;
13≥  5Z20;6;5;3;4;-8;-30;-4;-5;-10;11Z-1;3Z1;6Z1;-1;2Z  0;
14≥ 10Z20;6;5;3;4;-8;-30;-4;-5;-10;7Z-1;3Z1;6Z1;-1;1Z  0;
15≥ 15Z20;6;5;3;4;-8;-30;-4;-5;-10;3Z-1;3Z1;6Z1;-1;  0;
16≥     8;30;4;5;10;-5;-6;-30;-2;-3;15Z1;7Z-1;7Z  0;
17≥  5Z8;30;4;5;10;-5;-6;-30;-2;-3;11Z1;6Z1;-1;6Z  0;
18≥ 10Z8;30;4;5;10;-5;-6;-30;-2;-3;7Z1;6Z1;-1;5Z  0;
19≥ 15Z8;30;4;5;10;-5;-6;-30;-2;-3;3Z1;6Z1;-1;4Z  0;
20≤ 28;36;9;8;14;15;0;-25;1;1;15;0;-25;1;1;15;0;-25;1;1;
        -13;-36;-34;-7;-13;4Z4U3Z-1;3Z-1;  0;
```

Values of p_{ij} used in problems A to F.

The columns refer to the machines, the rows to the items to be processed. In the 4 and 5 item problems the first 4 and 5 in each table were used.

A

5	8	20
6	30	6
30	4	5
2	5	3
3	10	4
4	1	4

B

9	13	6
7	7	20
6	4	8
8	3	10
20	7	2
10	2	13

C

6	7	3
12	2	3
4	6	8
3	11	7
6	8	10
2	14	12

D

4	5	5
2	17	7
2	10	4
10	8	2
7	15	6
9	4	11

E

9	1	5
12	1	13
8	6	7
11	9	10
5	13	6
12	3	9

F

15	5	14
7	4	2
9	14	18
28	11	9
1	17	4
1	8	3

APPENDIX C

TABLES GIVING THE RESULTS OF RUNNING THE EXPERIMENTAL PROGRAMMES ON THE TEST DATA

Appendix C: Tables Giving the Results of Running the Experimental Programmes on the Test Data.

Explanation of the tables.

For a detailed account of the logic of the programmes as regards slacks and overflows the reader is referred to Parts 2 and 3 of Chapter 2. A brief explanation is given here.

Time: The time excludes the reading in of the data and printing out of the results but includes the monitoring printout which printed every value of D.

Pivots: This relates to the successful pivots and excludes attempts at pivoting which had to be backtracked because of integer overflow. "After rational solution" relates to the solution of the original linear programming problem except where stated in a footnote.

Cuts: These were constraints added with a non-zero constant term.

Slacks: These were Gomory type constraints with zero constant terms used for reducing the size of the determinant D. They also reduced the choice of cuts.

Overflows: These figures relate to the number of times a pivot operation had to be backtracked because of integer overflow.

Distance between integer and rational solutions:
This indicates the number of integer values the cost function had to pass through during solution. It is thought that this distance is one indication of the difficulty of the problem particularly in the case of programme BHD (see Chapter 3 Part 3). Where BGD is concerned the figure relates to the first artificial cost function (see Chapter 3 Part 3).

Value of D at rational solution:
This is the value of the determinant after solving the linear programming problem. Its size is an indication of the difficulty of solving the problem firstly as regards the problem of integer overflow and secondly as regards the choice of constraint.

Value of objective function:
This gives an indication of success where a programme produces approximate solutions. Where a programme was terminated prematurely it indicates how far it was from the solution to the problem.

The programmes:
These are described in Part 4 of Chapter 2. In the case of programme BGD the figures relate to the point at which the first integer solution was found, the point at which the last integer solution was found, and the point at which it was established that no more integer solutions existed and that the last one discovered was in fact the optimal one.

Problem 1:
Markowitz and Mann.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD· at 1st solution	14	28	16	4	3	2	0	0	0	319	504
at best, i.e. 2nd solution	168	332	320	31	63	62	19	19	1	319	540
at end of run.	238	506	494	43	89	88	23	23	2	319	540
BHD	1801*	3546	3529	426	572	571	239	237	42.4	25,619105	552
BHM	1801*	3549	3532	442	504	503	206	206	42.4	25,619105	552
BHP	1803*	1914	1897	633	567	566	437	437	19.5	25,619105	575
BHQ	1802*	2379	2362	657	857	856	170	170	23.8	25,619105	570.6
BH9	345+	426	409	149	118	117	17	17	15.5	25,619105	578.9
Cost function scaled by 7.5	399	693	680	79	96	96	34	34	3.9	176716	540
(using BHD)											

*run terminated by time limit.
+run terminated because integer overflow occurred when pivoting on a cut.

Problem 2: 2-dimensional knapsack. Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	10	17	5	1	5	4	0	0	0	18	135673
at best, i.e. 2nd solution	51	62	50	7	13	12	2	2	0	18	141278
at end of run	67	91	79	9	14	13	2	2	0	18	141278
BHD	36	45	27	6	15	15	0	0	741	40	141278
BHM	1805*	1907	1889	392	413	413	0	0	571.1	40	141447.9
BHP	330	378	360	75	78	78	0	0	741	40	141278
BHQ	1801*	1795	1777	356	558	558	43	43	567	40	141452

*run terminated by time limit.

Problem 3: 7-city travelling salesman / Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	246	170	129	9	23	23	0	0	0	35	277
at best, i.e. 3rd solution	666	461	420	17	79	79	4	4	1	35	267
at end of run.	727	514	473	19	82	82	4	4	1	35	267
BHD	552	418	379	22	59	59	1	1	39.1	35	267
BHM	570	420	381	24	58	58	1	1	39.1	35	267
BHP	507	401	362	18	52	52	1	1	39.1	35	267
BHQ	557	416	377	23	61	61	1	1	39.1	35	267

Problem 4: 7-city travelling salesman Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	68	54	16	2	3	2	0	0	0	14	267
at best, i.e. 1st solution	68	54	16	2	3	2	0	0	0	14	267
at end of run.	217	175	137	6	13	12	0	0	0	14	267
BHD	203	138	100	10	16	15	0	0	13.4	21	267
BHM	247	152	114	14	18	17	0	0	13.4	21	267
BHN	207	139	101	10	15	14	0	0	13.4	21	267
BHP	188	134	96	8	14	13	0	0	13.4	21	267
BHQ	217	145	107	12	18	17	0	0	13.4	21	267
BHE	250	157	119	16	15	14	0	0	13.4	21	267
BHF	339	183	145	27	22	21	0	0	13.4	21	267
BH6	*	111	83	14	27	27	0	0	1.4	343	255
BH9	202	136	98	10	15	14	0	0	13.4	21	267

*at this point BH6 started looping with a period of 38 pivots (2cuts, 26 slacks)

Problem 5: 6-city travelling salesman Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	38	50	16	3	2	1	0	0	0	20	63
at best, i.e. 1st solution	38	50	16	3	2	1	0	0	0	20	63
at end of run.	47	62	28	4	2	1	0	0	0	20	63
BHD	40	62	25	3	3	1	0	0	7.2	30	63
BHM	45	63	26	4	3	1	0	0	7.2	30	63
BHN	41	62	25	3	3	1	0	0	7.2	30	63
BHP	44	63	26	4	3	1	0	0	7.2	30	63
BHQ	40	62	25	3	3	1	0	0	7.2	30	63
BHE	41	62	25	3	3	1	0	0	7.2	30	63
BHF	46	65	28	4	4	2	0	0	7.2	30	63
BH6	34	50	16	3	3	1	0	0	7.2	6	63
BH9	41	62	25	3	3	1	0	0	7.2	30	63

Problem 6: Covering theorem.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD	*										
BHD	78	494	467	55	227	209	0	0	2.9	11	25
BHM	80	513	486	56	227	209	0	0	2.9	11	25
BHN	148	894	867	121	421	403	0	0	2.9	11	25
BHP	81	514	487	57	222	204	0	0	2.9	11	25
BHQ	127	762	735	100	385	367	0	0	2.9	11	25
BHE	228+	1077	1050	159	516	498	140	140	0.9	11	23
BHF	430+	2090	2063	370	907	889	214	214	0.9	11	23
BH6	ǂ	673	642	60	451	429	0	0	0.9	11	23
BH9	87	532	505	62	247	229	0	0	2.9	11	25

*as the objective function consisted entirely of ones BGD would have taken the same path as BHD

+at this point integer overflow occurred and as D was equal to 1 no cut could be added

ǂat this point BH6 started looping with a period of 18 pivots (2 cuts, 11 slacks)

Problem 7: A problem with large coefficients. Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	1	7	4	2	2	2	0	0	0	364807	40
at best, i.e. 2nd solution	3	20	17	6	8	7	0	0	1	364807	38
at end of run.	9	75	72	17	30	29	0	0	1	364807	38
BHD	13	129	126*	30	47	47*	4	3	6.6	*442,136027	38
BHM	13	112	109*	40	24	24*	5	4*	6.6*	*442,136027	38
BHN	52	381	378*	166	108	108*	3	2*	6.6*	*442,136027	38
BHP	38	318	315*	113	112	112*	3	2*	6.6*	*442,136027	38
BHQ	67	539	536*	201	224	224*	17	17*	6.6*	*442,136027	38
BHE	46	362	359*	164	110	110*	18	17*	6.6*	*442,136027	38
BHF	152	1086	1083*	451	379	379*	37	36*	6.6*	*442,136027	38
BH6	+	53	50*	21	13	13*	13	12*	3.6*	*442,136027	35
BH9	20‡	72	69*	44	16	16*	54	53*	2.6*	*442,136027	34.0

*the rational solution was never determined. These figures relate to the point at which the first cut was added.
+at this point BH6 started looping with a period of 11 pivots (4 cuts, 2 slacks).
‡at this point integer overflow occurred when pivoting on a cut.

Problem 8: A 2 x 2 problem. Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	0	4	2	1	1	1	0	0	0.5	11	455
at best, i.e. 1st solution	0	4	2	1	1	1	0	0	0.5	11	455
at end of run.	0	4	2	1	1	1	0	0	0.5	11	455
BHD	1	6	4	2	2	2	0	0	5	11	455
BHM	2	5	3	2	1	1	0	0	5	11	455
BHN	1	3	1	1	0	0	0	0	5	11	455
BHP	1	3	1	1	0	0	0	0	5	11	455
BHQ	1	4	2	1	1	1	0	0	5	11	455
BHE	1	8	6	3	3	3	0	0	5	11	455
BHF	0	4	2	1	1	1	0	0	5	11	455
BH6	0	4	2	1	0	0	0	0	5	11	455
BH9	1	6	4	2	2	2	0	0	5	11	455

Problem 9:
A 2 x 2 problem.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	1	7	4	2	2	1	0	0	0.5	11	445
at best, i.e. 1st solution	1	7	4	2	2	1	0	0	0.5	11	445
at end of run.	1	7	4	2	2	1	0	0	0.5	11	445
BHD	1	12	8	4	5	3	0	0	15	11	445
BHM	1	11	7	4	4	2	0	0	15	11	445
BHN	0	8	4	3	2	0	0	0	15	11	445
BHP	2	8	4	3	2	0	0	0	15	11	445
BHQ	0	11	7	3	5	3	0	0	15	11	445
BHE	2	18	14	7	7	5	0	0	15	11	445
BHF	2	11	7	3	5	3	0	0	15	11	445
BH6	1	11	7	3	5	3	0	0	15	11	445
BH9	1	14	10	4	6	4	0	0	15	11	445

Problem 10.1:
A product mix problem.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	2	9	5	2	5	2	0	0	0.5	2	5700
at best, i.e. 8th solution	25	132	128	29	51	48	1	1	2.5	2	8100
at end of run.	32	178	174	39	69	66	1	1	2.5	2	8100
BHD	7	59	42	8	39	26	0	0	76.5	17	8100
BHM	9	73	56	13	43	30	0	0	76.5	17	8100
BHN	5	46	29	7	28	15	0	0	76.5	17	8100
BHP	7	58	41	11	34	21	0	0	76.5	17	8100
BHQ	9	73	56	13	46	33	0	0	76.5	17	8100
BHE	9	76	59	13	51	38	0	0	76.5	17	8100
BHF	9	74	57	12	48	35	0	0	76.5	17	8100
BH6	15	125	108	17	88	75	0	0	76.5	17	8100
BH9	9	84	67	12	54	41	0	0	76.5	17	8100

Problem 10.2:
Problem 10.1 with cost function scaled.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	2	11	5	2	7	2	0	0	0.5	2	114
at best, i.e. 8th solution	14	120	114	21	57	52	0	0	2.5	2	162
at end of run.	16	137	131	23	67	62	0	0	2.5	2	162
BHD	5	44	32	7	26	18	0	0	1.5	17	162
BHM	5	43	31	7	25	17	0	0	1.5	17	162
BHN	4	37	25	6	21	13	0	0	1.5	17	162
BHP	6	48	36	10	26	18	0	0	1.5	17	162
BHQ	7	61	49	12	36	28	0	0	1.5	17	162
BHE	5	40	28	9	23	15	0	0	1.5	17	162
BHF	8	60	48	12	33	25	0	0	1.5	17	162
BH6	7	56	44	8	36	28	0	0	1.5	17	162
BH9	7	55	43	11	31	23	0	0	1.5	17	162

Problem 10.3:
Problem 10.1 with constraints sealed.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	2	9	5	2	5	2	0	0	0.5	2	5700
at best, i.e. 8th solution	18	98	94	28	25	22	0	0	2.5	2	8100
at end of run.	23	129	125	37	32	29	0	0	2.5	2	8100
BHD	3	16	10	6	4	2	0	0	76.5	17	8100
BHM	5	35	29	15	7	5	0	0	76.5	17	8100
BHN	2	10	4	3	2	0	0	0	76.5	17	8100
BHP	2	13	7	3	4	2	0	0	76.5	17	8100
BHQ	4	34	28	11	13	11	0	0	76.5	17	8100
BHE	5	32	26	10	12	10	0	0	76.5	17	8100
BHF	3	18	12	6	6	4	0	0	76.5	17	8100
BH6	4	25	18	7	8	5	0	0	76.5	17	8100
BH9	1	18	12	6	6	4	0	0	76.5	17	8100

Problem 10.4:
Problem 10.1 with all rows sealed.

Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	1	10	5	2	6	2	0	0	0.5	2	114
at best, i.e. 8th solution	13	82	77	21	23	19	0	0	2.5	2	162
at end of run.	14	92	87	23	24	20	0	0	2.5	2	162
BHD	2	21	16	7	5	4	0	0	1.5	17	162
BHM	3	20	15	7	4	3	0	0	1.5	17	162
BHN	1	9	4	3	1	0	0	0	1.5	17	162
BHP	2	11	6	3	2	1	0	0	1.5	17	162
BHQ	5	33	28	11	12	11	0	0	1.5	17	162
BHE	4	25	20	10	7	6	0	0	1.5	17	162
BHF	2	17	12	6	5	4	0	0	1.5	17	162
BH6	3	27	18	7	10	5	0	0	1.5	17	162
BH9	3	17	12	6	5	4	0	0	1.5	17	162

Problem A4: Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	15	32	13	3	3	2	0	0	0	27	22
at best, i.e. 1st solution	15	32	13	3	3	2	0	0	0	27	22
at end of run	21	48	29	4	4	3	0	0	0	27	22
BHD	33	83	54	5	11	8	0	0	1.9	27	22
BHM	34	87	58	6	7	4	0	0	1.9	27	22
BHP	64	115	86	18	13	10	0	0	1.9	27	22
BHQ	109	192	163	33	39	36	0	0	1.9	27	22
BHE	40	84	55	10	9	6	0	0	1.9	27	22
BHF	28	64	35	6	11	8	0	0	1.9	27	22
Problem scaled by 3 (using BHD)	22	54	28	3	10	6	0	0	1	9	23
Story and Wagner		261									

Problem B4: Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	12	25	5	2	3	2	0	0	0	80	10
at best, i.e. 1st solution	12	25	5	2	3	2	0	0	0	80	10
at end of run.	12	25	5	2	3	2	0	0	0	80	10
BHD	11	25	5	2	3	2	0	0	0	80	10
BHM	13	29	9	2	3	2	0	0	0	80	10
BHP	8	23	3	1	2	1	0	0	0	80	10
BHQ	9	24	4	1	3	2	0	0	0	80	10
BHE	14	30	10	3	2	1	0	0	0	80	10
BHF	9	24	4	1	3	2	0	0	0	80	10
Problem scaled by 3 (using BHD)	6	20	0	0	1	0	0	0	0	1	12
Story and Wagner		31									

Problem C4: Programme	Time (secs)	Pivots Total	 after rational solution	Cuts	Slacks Total	 after rational solution	Overflows Total	 after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	25	57	33	4	5	5	0	0	0	9	13
at best, i.e. 1st solution	25	57	33	4	5	5	0	0	0	9	13
at end of run.	27	64	40	4	5	5	0	0	0	9	13
BHD	30	70	45	6	6	5	0	0	1.4	25	13
BHM	39	88	63	7	8	7	0	0	1.4	25	13
BHP	39	88	63	7	8	7	0	0	1.4	25	13
BHQ	42	88	63	10	10	9	0	0	1.4	25	13
BHE	37	78	53	9	5	4	0	0	1.4	25	13
BHF	57	117	92	14	18	17	0	0	1.4	25	13
Problem scaled by 3 (using BHD)	11	27	4	2	1	1	0	0	0	2	13
Story and Wagner		59									

Problem D4: Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	6	21	0	0	0	0	0	0	0	1	28
at best, i.e. 2nd solution	9	24	3	1	0	0	0	0	0	1	26
at end of run.	10	25	4	1	0	0	0	0	0	1	26
BHD	6	22	0	0	0	0	0	0	0	1	26
BHM	6	22	0	0	0	0	0	0	0	1	26
BHP	7	22	0	0	0	0	0	0	0	1	26
BHQ	7	22	0	0	0	0	0	0	0	1	26
BHE	7	22	0	0	0	0	0	0	0	1	26
BHF	6	22	0	0	0	0	0	0	0	1	26
Problem scaled by 3 (using BHD)	8	25	0	0	0	0	0	0	0	1	28
Story and Wagner		37									

Problem E4: Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	45	76	51	10	10	6	0	0	0	19	23
at best, i.e. 2nd solution	120	209	184	27	16	12	0	0	0	19	19
at end of run.	154	292	267	32	18	14	0	0	0	19	19
BHD	139	306	277	27	39	35	0	0	4.4	317	19
BHM	139	300	271	29	31	27	0	0	4.4	317	19
BHP		281	252	30	34	30	0	0	4.4	317	19
BHQ	226	447	418	50	74	70	0	0	4.4	317	19
BHE	206	402	373	49	54	50	0	0	4.4	317	19
BHF	254	491	462	59	87	83	0	0	4.4	317	19
Problem scaled by 3 (using BHD)	39	75	50	9	9	9	0	0	1.4	88	19
Story and Wagner		261									

Problem F4 / Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution at best, i.e. 3rd solution at end of run.	8 56 57	27 90 94	0 63 67	0 14 14	0 8 8	0 8 8	0 0 0	0 0 0	0 0 0	1 1 1	35 30 30
BHD	61	139	106	12	17	15	0	0	5.2	203	30
BHM	69	156	123	14	14	12	0	0	5.2	203	30
BHP		128	95	11	10	8	0	0	5.2	203	30
BHQ	72	140	107	20	22	20	0	0	5.2	203	30
BHE	47	101	68	10	14	12	0	0	5.2	203	30
BHF	63	122	89	16	20	18	0	0	5.2	203	30
Problem scaled by 3 (using BHD)	29	62	35	7	4	3	0	0	2.7	30	31
Story and Wagner		62									

Problem A5 Programme	Time (secs)	Pivots Total	 after rational solution	Cuts	Slacks Total	 after rational solution	Overflows Total	 after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution at best, i.e. 1st solution at end of run.	19 19 21	36 36 40	0 0 4	0 0 0	1 1 1	0 0 0	0 0 0	0 0 0	0 0 0	1 1 1	25 25 25
BHD	17	35	0	0	1	0	0	0	0	1	25
BHM	17	35	0	0	1	0	0	0	0	1	25
BHP	18	35	0	0	1	0	0	0	0	1	25
BHQ	16	35	0	0	1	0	0	0	0	1	25
Problem sealed by 3 (using BHD)	21	42	0	0	1	0	0	0	0	1	25
Story and Wagner		613									

Problem B5 / Programme	Time (secs)	Pivots		Cuts	Slacks		Overflows		distance between integer and rational solutions	value of D at rational solution	value of objective function
		Total	after rational solution		Total	after rational solution	Total	after rational solution			
BGD: at 1st solution	26	45	9	1	1	1	0	0	0	88	13
at best, i.e. 1st solution	26	45	9	1	1	1	0	0	0	88	13
at end of run.	26	45	9	1	1	1	0	0	0	88	13
BHD	45	78	42	3	3	2	0	0	0	352	13
BHM	44	70	34	3	1	0	0	0	0	352	13
BHP	44	76	40	3	1	0	0	0	0	352	13
BHQ	51	79	43'	4	4	3	0	0	0	352	13
Problem scaled by 3 (using BHD)	32	61	21	1	2	1	0	0	0	4	14
Story and Wagner		71									

Problem C5 / Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	114	145	114	8	17	15	0	0	0	93	11
at best, i.e. 1st solution	114	145	114	8	17	15	0	0	0	93	11
at end of run.	114	145	114	8	17	15	0	0	0	93	11
BHD	106	145	114	8	17	15	0	0	1	93	11
BHM	110	148	117	10	13	11	0	0	1	93	11
BHP	127	160	129	13	14	13	0	0	1	93	11
BHQ	144	166	135	17	24	22	0	0	1	93	11
Problem scaled by 3 (using BHD)	50	61	29	6	5	4	0	0	0	2	11
Story and Wagner		141									

Problem D5 / Programme	Time (secs)	Pivots: Total	Pivots: after rational solution	Cuts	Slacks: Total	Slacks: after rational solution	Overflows: Total	Overflows: after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution	19	37	0	0	0	0	0	0	0	1	37
at best, i.e. 2nd solution	23	39	2	1	0	0	0	0	0	1	35
at end of run.	23	40	3	1	0	0	0	0	0	1	35
BHD	21	45	0	0	0	0	0	0	0	1	35
BHM	20	45	0	0	0	0	0	0	0	1	35
BHP	20	45	0	0	0	0	0	0	0	1	35
BHQ	20	45	0	0	0	0	0	0	0	1	35
Problem scaled by 3 (using BHD)	18	38	0	0	0	0	0	0	0	1	37
Story and Wagner		46									

Problem E5 Programme	Time (secs)	Pivots Total	 after rational solution	Cuts	Slacks Total	 after rational solution	Overflows Total	 after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution at best, i.e. 3rd solution at end of run.	188 761 761	256 966 966	220 930 930	13 57 57	17 82 82	15 80 80	0 0 0	0 0 0	0 1 1	144 144 144	22 18 18
BHD	469	673	620	37	52	50	0	0	3.5	2250	18
BHM	508	724	671	39	61	59	0	0	3.5	2250	18
BHP	591	809	756	50	69	67	0	0	3.5	2250	18
BHQ	862	1062	1009	90	129	127	0	0	3.5	2250	18
Problem scaled by 3 (using BHD)	101	126	91	11	15	14	0	0	1.3	104	20
Story and Wagner		*1000									

*run terminated by limit on the number of pivots.

Problem F5 Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BGD: at 1st solution at best, i.e. 2nd solution at end of run.	57 191 233	78 224 284	38 184 244	4 19 22	4 11 13	3 10 12	0 0 0	0 0 0	0 0 0	4 4 4	33 28 28
BHD	348	484	433	30	50	45	0	0	6.2	4646	28
BHM	326	467	416	28	42	37	0	0	6.2	4646	28
BHP	288	361	310	32	39	34	0	0	6.2	4646	28
BHQ	451	499	448	53	82	77	0	0	6.2	4646	28
Problem scaled by 3 (using BHD)	129	169	132	11	24	24	0	0	2.6	12750	28
Story and Wagner		323									

Problem A6 Programme	Time (secs)	Pivots		Cuts	Slacks		Overflows		distance between integer and rational solutions	value of D at rational solution	value of objective function
		Total	after rational solution		Total	after rational solution	Total	after rational solution			
BHD	838	408	357	9	160	146	46	42	* 16.8	*396,232673	22
Problem scaled by 3 (using BHD)	90	105	21	2	3	2	0	0	0	30	23
Story and Wagner		807									

Problem B6 Programme	Time (secs)	Pivots Total	 after rational solution	Cuts	Slacks Total	 after rational solution	Overflows Total	 after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BHD	63	65	22	2	5	4	0	0	0	440	10
Problem scaled by 3 (using BHD)	52	68	0	0	2	0	0	0	0	1	12
Story and Wagner		83									

Problem C6 Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BHD	605	463	420	32	65	61	0	0	2.5	1440	14
Problem sealed by 3 (using BHD)	111	96	48	6	7	5	0	0	0	2	14
Story and Wagner		891									

Problem D6 Programme	Time (secs)	Pivots		Cuts	Slacks		Overflows		distance between integer and rational solutions	value of D at rational solution	value of objective function
		Total	after rational solution		Total	after rational solution	Total	after rational solution			
BHD	51	61	59	1	1	1	0	0	0	5	28
Problem scaled by 3 (using BHD)	46	55	1	1	0	0	0	0	0	2	30
Story and Wagner		55									

Problem E6 / Programme	Time (secs)	Pivots		Cuts	Slacks		Overflows		distance between integer and rational solutions	value of D at rational solution	value of objective function
		Total	after rational solution		Total	after rational solution	Total	after rational solution			
BHD	919	823	755	38	80	79	0	0	3.04	30000	18
Problem scaled by 3 (using BHD)	77	74	8	3	4	3	0	0	0.5	8	20
Story and Wagner		*1000									

*run terminated by limit on the number of pivots.

Problem F6 Programme	Time (secs)	Pivots Total	Pivots after rational solution	Cuts	Slacks Total	Slacks after rational solution	Overflows Total	Overflows after rational solution	distance between integer and rational solutions	value of D at rational solution	value of objective function
BHD	*1867	1384	+1313	63	192	+177	45	+39	+6.2	+11728198860	24
Problem scaled by 3 (using BHD)	798	734	672	35	56	45	0	0	2.5	1440	26
Story and Wagner		1000*									

*run terminated by limit on time on the number of pivots.
+the rational solution was never determined. These figures relate to the point at which the first cut was added.

APPENDIX D

THE EXPERIMENTAL PROGRAMMES

PART 1

THE TWO MOST EFFECTIVE PROGRAMMES

Appendix D : The Experimental Programmes.

Part 1 : The Two Most Effective Algorithms.

Specification of Programme BHD.

Purpose.

The programme minimises a function $\sum_{j=1}^{n} c_j x_j$ subject to the constraints $\sum_{j=1}^{n} a_{ij} x_{ij} \substack{\le \\ = \\ \ge} b_i$ (i=1,...m)

all the x_j being constrained to be non-negative integers.

Data.

The layout of the numerical data is as follows, the letters having the same meaning as in the previous section:

↑ a title consisting of the identifier of the user plus any sequence of algol basic symbols excluding '↑'↑

m;	n;						
0	;	c_1 ;	c_2 ;	......	c_n ;		
1	ρ_1	a_{11} ;	a_{12} ;	......	a_{1n} ;	b_1	;
2	ρ_2	a_{21} ;	a_{22} ;	......	a_{2n} ;	b_2	;
		.	.	.			
m	ρ_m	a_{m1} ;	a_{m2} ;	......	a_{mn} ;	b_m	;

The ρ_i must be the terminators of the previous number and are either $\le$, $=$, or $\ge$ according to the relation $\sum_{j=1}^{n} a_{ij} x_j \rho_i b_i$,

e.g. the row

$$3 \ge 1;\ 1;\ 4;$$

represents the third constraint $x_1 + x_2 \ge 4$. Each a_{ij}, b_i and c_j must be an integer.

Although the programme will accept data regardless of the signs of the a_{ij}, b_i and c_j and the values of the ρ_i, the user is advised to restrict his data to one of the following forms:

either $b_i \ge 0$, ρ_i being = or $\le$

or $c_j \ge 0$, ρ_i being $\ge$ or $\le$

With other configurations the programme may terminate erroneously.

The above form of data may be repeated as many times as desired, and the programme is terminated by a nominal set of data as follows:

↑ ↑

-1;→

Method.

The programme is one of the many variants of Gomory's Method of Integer Forms (ref 1). It is described in Part 3 and Part 4 (a) of Chapter 2.

Output.

The programme produces a considerable amount of output to enable the user to monitor the progress of the programme towards a solution.

Every pivot element is printed out. They are printed sequentially, six to a line. If three asterisks are printed it indicates that integer overflow occurred and that the last figure printed represents an unsuccessful attempt at pivoting. If a pivot is followed by an S it indicates that a constraint with zero constant term has been added to scale the tableau. If a C is printed it indicates a cut has been added.

The more important monitoring information is

(a) The rational solution is printed giving the value of every basic Variable.
(b) Every time a cut is added the number of the interation and the value of the cost function is printed provided this value has changed since the last iteration.
(c) When an integer solution is found the values of the basic variables are printed followed by the entire array.

The following error messages may be encountered:

SOLUTION UNBOUNDED:
The solution to the original linear programming problem is unbounded.

LEXICOGRAPHICALLY UNBOUNDED:
Although the problem itself is bounded one of the variables is not. The user must add a constraint giving this variable a bound.

INTEGER OVERFLOW:
Integer overflow has occurred in circumstances with which the programme is unable to cope.

NO RATIONAL SOLUTION (P):
It has not been possible to eliminate the artificial variables.

NO RATIONAL SOLUTION (D):
The problem was discovered to be infeasible while performing the Dual Simplex Algorithm. If the rational solution has already been found this message is to be interpreted as meaning there is no integer solution.

LOGICAL ERROR:
This should never occur. It could arise from a number of places and means a programme error, an undetected error in the format of the data, or a machine fault.

NO INTEGER SOLUTION:
A basic variable has non-integer constant term but the coefficients of the associated row have all integer values.

Computer requirements.

Paper tape reader
Line printer
Core store: programme plus (m+n+6) x (n+2) + 50 words where m and n are as previously defined.

Specification of Programme BGD.

This programme is the same as BHD save for one statement which is contained in the comment on page 188. Its specification is the same save for the following points.

Method.

The programme approaches the optimal integer solution via a series of approximate ones and is described in Part 3 and Part 4(j) of Chapter 2.

Output.

This is the same as that of Programme BHD except that after an integer solution is printed the solution process continues and with it the monitoring printout. The programme terminates computation with the message 'NO BETTER SOLUTION' and the last integer solution printed is the optimal one.

Computer requirements.

An extra $s(n+1)$ words of store is needed where s is the smallest number such that 2^s is not less than the largest element of the objective function, i.e. $2^s > \max(\mathrm{abs}(c_j))$, $j=1,\ldots,n$.

```
begin comment This is the text of programme BHD.  Where it differs
          from the text of the other programmes is indicated in the
          comments;
     library A6,A12,A13,A14;
     comment The library functions used were those that dealt with
          input-output.  Only three of the standard functions were
          used and the KDF9 User Code versions of these follow;
       real procedure abs(x); value x; real x;
       KDF9 1/0/0/0;
       [x]; ABSF; EXIT; ALGOL;
       integer procedure sign(x); value x; real x;
       KDF9 2/0/0/0;
       [x]; ZERO; SIGNF; EXIT; ALGOL;
       integer procedure entier(x); value x; real x;
       KDF9 4/0/0/1;
       V0=B4322506316227052;
       V1=B1571640000000000;
       [x]; FIX; DUP; SET39; -; DUP; J3>Z;
       =C3; J1>Z;
       J2<Z;
       ZERO; EXIT;
2;     SET-1; EXIT;
1;     SET-8; =+C3; SHLC3; NC3; SHLC3;
       NC3; SHAC3; EXIT;
3;     ERASE; FLOAT; SETAV0; REV; SET1; JSP299;
       ALGOL;
```

```
procedure printar(a,m,n,p,q,D,g); value m,n,D;
integer m,n,D,g; integer array a,p,q;
comment This procedure prints the contents of arrays a, p, and q
  arranged as a matrix. It chooses a format to fit the largest
  element of a and the value of this format is assigned to parameter
  g. The arrays are dimensioned a[-1:m,0:n], p[1:m], q[1:n];
begin integer i,j,I,J,s,f;
        I:=min(-abs(a[i,min(-abs(a[i,j]),j,0,n,true)]),i,-1,m,true);
        J:=min(-abs(a[I,j]),j,0,n,true);
        s:=if abs(a[I,J])>D then abs(a[I,J]) else D;
        f:=if s<1000 then format([-ndd;])
             else if s<10^6 then format([-nddddd;])
                else if s<10^9 then format([-nddsdddddd;])
                  else format([-ndddddsdddddd;]);
        writetext(30,[D=]); write(30,f,D); newline(30,2);
        for i:=-1 step 1 until m do
          begin write(30,f,if i>0 then p[i] else i);
                for j:=0 step 1 until n do write(30,f,a[i,j]);
                newline(30,1+(i+1)÷(m+1))
          end;
        write(30,f,0); write(30,f,0);
        for j:=1 step 1 until n do write(30,f,q[j]); newline(30,3);
        g:=f;
end;
integer procedure time;
comment This procedure assigns to time the run time used so far by
  this programme rounded down to the nearest second;
KDF9 1/0/0/0;
        SET3; OUT; SHL-24; EXIT;
ALGOL;
```

```
integer procedure hcf(a,b); value a,b; integer a,b;
comment This procedure assigns to hcf the highest common
  factor of a and b;
begin   a:=abs(a); b:=abs(b);
        if a=0 or b=0 then goto H; if a<b then goto B;
A:      a:=a-a÷b×b; if a=0 then goto H;
B:      b:=b-b÷a×a; if b≠0 then goto A;
H:      hcf:=if a=0 then b else a
end;
integer procedure euclidalg(h,D); value h,D; integer h,D;
comment This procedure assigns to euclidalg a number
  between 0 and D such that
                euclidalg×h ≡ hcf(h,D)   (mod D);
begin integer k,u,v,g;
        h:=dlrem(h,1,D); k:=D; u:=1; v:=0; if h=0 then goto E;
G:      g:=k÷h; k:=k-g×h; v:=v-g×u; if k=0 then goto E;
        g:=h÷k; h:=h-g×k; u:=u-g×v; if h≠0 then goto G;
E:      euclidalg:=if k=0 then u else u+v
end;
integer procedure min(t,s,p,q,B); real t; integer s,p,q; boolean B;
comment This procedure uses Jensen's device to find the minimum of a
  one dimensional array subject to a boolean expression;
begin real z;
        min:=0; z:=₁₀11;
        for s:=p step 1 until q do
          if B then begin if t<z then begin min:=s;
                                             z:=t
end                 end               end;
```

```
procedure Intch(a,m,n,p,q,D,I,J,FAIL,oflow); value m,n,I,J;
integer m,n,D,I,J; integer array a,p,q; switch FAIL; label oflow;
comment This procedure performs a pivot operation on a[I,J]. In the
  event of overflow the array a is restored to its original state and
  the procedure exits to label oflow.
        Apart from I,J, and oflow, the parameters perform the same
  function as the variables with the same identifiers described at
  the start of the main programme;
begin integer i,j,D1,D2,g;
        write(30,format([-nddddddsdddddds]),a[I,J]);
        D2:=a[I,J]; D1:=D×sign(D2);
        trans(a,m,n,D1,D2,I,J,i,j,LOFLOW,FAIL[5]);
        if D2<0 then
          begin for j:= 0 step 1 until n do if j≠J then a[I,j]:=-a[I,j] end
           else for i:=-1 step 1 until m do if i≠I then a[i,J]:=-a[i,J];
        a[I,J]:=D1; D:=abs(D2); g:=p[I]; p[I]:=q[J]; q[J]:=g;
        goto DONE;
LOFLOW: for j:=j-1 step -1 until 0 do if j≠J then
          a[i,j]:=dlprod(D1,a[i,j],a[I,j],a[i,J],D2,FAIL[5],FAIL[5]);
        for i:=i-1 step -1 until -1 do if i≠I then
          begin for j:=n step -1 until J+1,J-1 step -1 until 0 do
                  a[i,j]:=dlprod(D1,a[i,j],a[I,j],a[i,J],D2,FAIL[5],FAIL[5])
          end;
        writetext(30,[XX[16s]X*]); goto oflow;
DONE:
end;
```

```
procedure trans(a,m,n,D1,D2,I,J,io,jo,OFLOW,ERROR); value m,n,D1,D2,I,J;
integer array a; integer m,n,D1,D2,I,J,io,jo; label OFLOW,ERROR;
comment The purpose of this procedure is to perform that part of the
  pivot operation which replaces each element, a[i,j], of a by
  (D2×a[i,j]-a[i,J]×a[I,j])÷D1, where I and J are the pivot row and
  column, respectively.  It does not alter the pivot row and column
  themselves.
        The advantages of writing this procedure in User Code are
   (i)   it permits the use of double length arithmetic,
   (ii)  it is easier to detect overflow without terminating the
         programme,
   (iii) it speeds up a procedure in which the programme spends a
         large proportion of its time.

   The integer array a is dimensioned a[-1:m,0:n],
   D1 and D2 are the old and new values of the determinant
  multiplied by the sign of the pivot element,
   I is the pivot row,
   J is the pivot column,
   io and jo are only defined if overflow occurs: they are such that
  a[io,jo] is the element on which overflow occurred,
   OFLOW is the error exit if overflow occurs,
   ERROR is the error exit if the division leaves a remainder:
  its occurrence indicates either a logical error in the programme
  or a machine fault;
```

```
comment The Algol equivalent of this procedure is as follows:
    for io:= -1 step 1 until I-1,I+1 step 1 until m do
      for jo:=  0 step 1 until J-1,J+1 step 1 until n do
        a[io,jo]:=d1prod(D2,a[io,jo],-a[I,jo],a[io,J],D1,OFLOW,ERROR);
KDF9 6/6/2/0;
        [a]; SHC-16; =Q11; [m]; [I]; DUP; NOT; NEG; =RC14; DUP;
        SET AYO; C11; +; DUP; =M10; DUP; NEG; NOT; =M14; +; =M12; -; =I13;
        Y1M11; SETB 177777; AND; [J]; DUPD; =C12; =I15; XD; CONT; =M13;
        [n]; C12; -; =C13; ZERO; [D1]; DUP; ABS; =Q11; J99>Z; NOT;  99; =C10;
        (Q10= 0 or -1/ - /Aa[0,0]);      (Q11=       abs(D1)      );
        (Q12=       J/ - /Aa[I,0]);      (Q13=     n-J/m-I/JX(L+3));
        (Q14= row ctr/ 1 /Aa[i,0]);      (Q15= col ctr/L+3/jX(L+3));
        [D2]; J17C10Z; NEG;  17; SHA+8; JS3; I13; =C14; JS4; ERASE; EXIT;
*3;     M0 TO Q15; C12 TO Q15; M13M14; DUP; J10=Z; J18C10Z; NEG;  18; SHA+8;
        REV; JS5; C13 TO Q15; M+I15; JS5;
12;     REV; ERASE; DC14;
 4;     M+I14; J3C14NZ; EXIT 1;
*1;     DUPD; M14M15; XD; CAB; M12M15; XD; -D;
11;     Q11; +R; SHA-8; J2V; =M14M15Q; J6≠Z;
 5;     J1C15NZ; EXIT 1;
10;     ERASE; Q11; REV; JS7; C13 TO Q15; M+I15; JS7; J12;
*8;     DUPD; M14M15; XD;
 9;     CAB; +R; SHA-8; J2V; =M14M15Q; J6≠Z;
 7;     J8C15NZ; EXIT 1;
 2;     LINK; LINK;  20; ERASE; J20NEN;
        M14; M10; -; =[io]; M15; I15; +I; ERASE; =[jo]; J[OFLOW];
 6;     LINK; LINK;  60; ERASE; J60NEN; J[ERROR];
ALGOL;
```

```
integer procedure dlprod(a,b,c,d,e,oflow,error); value a,b,c,d,e;
integer a,b,c,d,e; label oflow,error;
comment This procedure is normally equivalent to
          dlprod:=(a×b+c×d)÷e.
  However, it performs the arithmetic in double length and in the
  event of the result overflowing or the division leaving a
  remainder it exits to 'oflow' or 'error' respectively;
KDF9 4/0/0/0;
        [a]; [b]; XD; [c]; [d]; XD; +D; [e]; DUP; J1>Z;
        PERM; NEGD; CAB; NEG;
1;      ÷R; SHA+8; SHA-8; J2V; REV; J3≠Z; EXIT;
2;      ERASE; ERASE; J[oflow];
3;      ERASE; J[error];
ALGOL;
integer procedure dlsign(a,b,c,d); value a,b,c,d; integer a,b,c,d;
comment This procedure is normally equivalent to
          dlsign:=sign(a×b-c×d).
  It performs the arithmetic in double length to avoid the
  possibility of overflow;
KDF9 4/0/0/0;
        [a]; [b]; XD; [c]; [d]; XD; -D; OR; ZERO; SIGN; EXIT;
ALGOL;
integer procedure dlrem(a,g,D); value a,g,D; integer a,g,D;
comment This procedure is normally equivalent to
          dlrem:=a×g-a×g÷D×D.
  It performs the arithmetic in double length to avoid the
  possibility of overflow.  It assumes that D is positive;
KDF9 3/0/0/0;
        [a]; [g]; XD; [D]; ÷R; ERASE; EXIT;
ALGOL;
```

integer procedure pivot(a,m,n,I,i0,FAIL); **value** m,n,I,i0;

integer array a; **integer** m,n,I,i0; **label** FAIL;

comment Given the pivot row, I, this procedure selects the pivot column according to the rules for the lexicographic Dual Simplex Method and assigns it to pivot. The Dual Simplex Method selects a pivot column J such that a[I,J]<0 and a[i0,J]/abs(a[I,J]) is a minimum over J, where i0 is the cost row. The lexicographic rule lays down that in the event of a tie between two rows the ratios a[i0+1,J]/abs(a[I,J]) are compared, and so on until the tie is resolved. The procedure may also be used to find the pivot column in the Simplex Method by inserting a dummy pivot row consisting entirely of -1's.

The reason for writing this procedure in User Code was that in some problems columns appeared with large numbers of zeros at the top of them, with the result that the number of operations needed to choose a pivot column was of the order of mXn, rather than simply n.

Array a is dimensioned a[-1:m,0:n],

I is the pivot row,

i0 is the first cost function: -1 when called by Intsimp, 0 when called by Dintsimp,

FAIL is the error exit if no feasible pivot column can be found.

The Algol equivalent of this procedure is contained in **integer procedure** pivot2, on the following page.

The procedure was altered in programme BH6 to omit the lexicographic rule for breaking ties;

```
KDF9 4/7/0/0;

    [a]; SHC-16; =Q11; C11; SETAY0; +; Y1M11; DUP; =I15; +; =M15;
    [i0]; DUP; =RM10; [m]; NOT; NEG; REV; -; DUP; =C10; =C9;
    [n]; DUP; =C15; =C12; [I]; =M12; ZERO; =C14; SET-1; =M11;
```

```
*1;     M12M15; J9≥Z; M11M15; J9>Z; Q10TOQ13;
*2;     M15M13Q; J3≠Z; *J2C13NZS; J9;
*3;     M-I13; M15M13; ZERO; SIGN; NEG; NOT; =C11; M13; DUP; =I11; J7C14Z;
        I12; -; DUP; J7>Z; J4=Z; J9C11Z; ZERO; J7;
*4;     C9; M13; -; =C13; M14M13; M15M12; XD; M14M12; M15M13Q;
        XD; -D; OR; DUP; J7<Z; J9>Z; J4C13NZ; J9;
 7;     ERASE; Q15TOQ14; I11TOQ12; J9C11Z; I11; NOT; NEG; M10; -; =C10;
 9;     M+I15; DC15; J1C15NZ; J10C14Z; C12; C14; -; NOT; NEG; EXIT;
10;     J[FAIL];
ALGOL;
integer procedure pivot2(a,m,n,I,i0,FAIL); value m,n,I,i0;
integer array a; integer m,n,I,i0; label FAIL;
comment Although not called by the programme this procedure has been
  inserted here because it contains the Algol equivalent of integer
  procedure pivot;
begin integer J,i,j,g;
        J:=0;
        for j:=1 step 1 until n do if a[I,j]<0 and a[-1,j]≤0 then
          begin if J=0 then goto FND
                for i:=i0 step 1 until m do
                  begin g:=dlsign(a[i,j],a[I,J],a[I,j],a[i,J]);
                        if g>0 then goto FND; if g<0 then goto NEXT
                  end;
                goto NEXT;
          FND:  J:=j;
          NEXT:
          end;
        pivot2:=J; if J=0 then goto FAIL
end;
```

```
procedure Intsimp(a,m,n,p,q,D,R,L,FAIL,x,z); value n,x;
integer m,n,D,R,L,x; integer array a,p,q,z; switch FAIL;
comment This procedure performs the Simplex algorithm. It produces a
  lexicographically optimal tableau, i.e. the first non-zero element
  of every column is positive. The purpose of this is to assist the
  selection of constraints in the integer programming part of the
  programme, but it also enables the artificial cost, in row -1, and
  the objective function, in row 0, to be optimised simultaneously.
        In the event of overflow procedure Scale is called. If
  successful, another attempt is made at pivoting, if unsuccessful,
  the run is abandoned.
        The parameters perform the same function as the variables with
  the same identifiers described at the start of the main programme;
begin integer i,j,I,J; boolean success;
ONE:    for j:=1 step 1 until n do a[m+1,j]:=-1;
TWO:    J:=pivot(a,m,n,m+1,-1,FOUR); i:=-2;
THREE:  i:=i+1; if a[i,J]=0 and i<m then goto THREE
                  else if a[i,J]≥0 then goto FOUR;
        I:=min(a[i,0]/a[i,J],i,z[10]+1,m,a[i,J]>0);
        if I=0 then goto FAIL[if a[0,J]<0 then 1 else 7];
        p[m+1]:=p[I]; for j:=0 step 1 until n do a[m+1,j]:=a[I,j];
        space(30,5);
        Intch(a,m+1,n,p,q,D,m+1,J,FAIL,OFLOW); goto ONE;
OFLOW:  Scale(a,m,n,p,q,D,R,L,FAIL,success);
        goto if success then ONE else FAIL[6];
FOUR:   for i:=1 step 1 until m do if p[i]<0 and a[i,0]>0 then goto FAIL[3]
end;
```

```
procedure Dintsimp(a,m,n,p,q,D,R,L,FAIL,x,z); value n,x;
integer m,n,D,R,L,x; integer array a,p,q,z; switch FAIL;
comment This procedure performs the Dual Simplex algorithm. It chooses
  the row with least a[i,0] to pivot on, calls integer procedure pivot
  to locate the pivot column, and calls procedure Intch to effect the
  transformation. In the event of overflow occurring in Intch
  procedure Scale is called and if successful another attempt is made
  to choose a pivot element and pivot on it successfully. If no
  overflow occurs procedure Scale is called nonetheless before exiting
  from the procedure.
        The parameters perform the same function as the variables with
  the same identifiers described at the start of the main programme;
begin integer i,I,J,g; boolean success;
ROW:    g:=I:=0;
        for i:=z[10]+1 step 1 until m do if a[i,0]<g then
          begin I:=i; g:=a[i,0] end;
        if g=0 then goto DONE;
        J:=pivot(a,m,n,I,0,FAIL[2]);
        p[m+1]:=p[I]; for j:=0 step 1 until n do a[m+1,j]:=a[I,j];
        space(30,5);
        Intch(a,m+1,n,p,q,D,m+1,J,FAIL,OFLOW); goto ROW;
OFLOW:  Scale(a,m,n,p,q,D,R,L,FAIL,success);
        if success then goto ROW;
        writetext(30,[[13s]NON-OPT]); goto FIN;
DONE:   Scale(a,m,n,p,q,D,R,L,FAIL,success);
FIN:
end;
```

```
procedure Scale(a,m,n,p,q,D,R,L,FAIL,success); value m,n,R,L;
integer m,n,D,R,L; integer array a,p,q; switch FAIL; boolean success;
comment The purpose of this procedure is to search for 'constraints'
  with zero constant term and incorporate them into the tableau by
  means of a pivot operation. This has the effect of reducing the
  value of D without altering the value of any a[i,0]/D.
        The advantage of adding such constraints is that the value
  of D is reduced while maintaining optimality and feasibility.
        The parameter 'success' is assigned the value true if at
  least one such constraint is found, false otherwise. The other
  parameters perform the same function as those with the same
  identifiers described at the start of the main programme;
begin integer i,j,k;
        success:=false;
AGAIN:  for i:=0 step 1 until m do
          begin k:=D÷hcf(D,a[i,0]);
                if k≠D then for j:=1 step 1 until n do
                  begin if dlrem(a[i,j],k,D)≠0 and a[-1,j]≤0
                                then goto FOUND end
          end;
        goto DONE;
FOUND:  success:=true; m:=m+1; p[m]:=999; writetext(30,[S****]);
        for j:=0 step 1 until n do a[m,j]:=-dlrem(a[i,j],k,D);
        Intch(a,m,n,p,q,D,m,pivot(a,m,n,m,0,FAIL[5]),FAIL,FAIL[6]);
        m:=m-1; goto AGAIN;
DONE:
end;
```

```
procedure Integer(a,m,n,p,q,D,R,L,FAIL,x,z);
integer m,n,D,R,L,x; integer array a,p,q,z; switch FAIL;
comment The purpose of this procedure is to choose a new constraint,
  add it to the tableau, and restore the tableau to feasibility.
  This is done by calling procedures Constraint, Intch and Dintsimp.
  The procedure also checks the time taken so far and prints certain
  monitoring information.
        The functions of the parameters are the same as those with
  the same identifiers described at the start of the main programme;
begin integer f,fi,fr,pa0,pD,tm,t; boolean finish;
        f:=format([-ndddddsdddddds]); fr:=format([+d.ddddsddddsdddd⏨+nd]);
        fi:=format([sssss-ndddddsssss]);
        tm:=time; t:=0; finish:=false; pa0:=a[0,0]+1; pD:=D;
REPORT: if z[7]+time-tm>1800 then finish:=true;
        if finish or dlsign(pa0,D,a[0,0],pD)≠0 then
          begin pa0:=a[0,0]; pD:=D; newline(30,1);
                write(30,fi,t); write(30,fr,pa0/D); space(30,6);
                write(30,f,pa0); writetext(30,[/]); write(30,f,pD);
                if finish then goto DONE
          end;
        Constraint(a,m,n,p,D,FAIL[4],finish);
        if finish then goto REPORT; t:=t+1; p[m+1]:=R+t;
        Intch(a,m+1,n,p,q,D,m+1,pivot(a,m+1,n,m+1,0,FAIL[4]),FAIL,FAIL[6]);
        Dintsimp(a,m+1,n,p,q,D,R,L,FAIL,x,z); goto REPORT;
DONE:   newline(30,1)
end;
```

```
procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment This procedure generates and adds a constraint according to
  the particular algorithm being tested. The majority of programmes
  differed from BHD only in this procedure, the exceptions being BGD,
  which has one extra statement in the main programme, and BH6, which
  differs in integer procedure pivot. The text of the procedure
  reproduced here is common to programmes BHD and BGD.
        Programmes BHD and BGD choose the first row with an a[i,0]/D
  which is non-integer, calculate dlrem(a[i,0],1,D), that is the
  remainder af a[i,0] when divided by D, and multiply the row by the
  largest multiple of the remainder which is less than D, and take
  the constraint from this row.
        The parameter fail is not used in this version of the
  procedure, finish is set to true if the current solution is an
  integer one, and the remaining parameters perform the same function
  as the variables with the same identifiers described at the start of
  the main programme;
begin integer i,j,g;
        for i:=0 step 1 until m do if dlrem(a[i,0],1,D)≠0 then goto FND;
        finish:=true; goto DONE;
FND:    g:=(D-1)÷dlrem(a[i,0],1,D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;
```

comment This is the start of the main programme. The programme first reads the dimensions of the data, m and n, followed by the variable parameters whose presence is indicated by n<0. The parameters are held in integer array z.

The objective function is read and temporarily placed in integer array q. The number of cost rows to be generated is calculated and assigned to s. In all but programme BGD s=0: in programme BGD it is such that 2↑s is the largest power of 2 less than or equal to the coefficient of the objective function with maximum absolute value.

Variable m is now increased, viz. m:=m+n+s, and integer array a is declared to be large enough to hold a problem with this number of rows. The s+1 cost functions are now generated. Next the constraints on the original data are read and assigned to rows a[s+1,j] to a[m,j], where j=1,..,n, but leaving rows a[s+y+1,j] to a[s+y+n,j] free for the negative unit matrix which is next to be generated. y signifies the number of constraints of the original data to be placed above the negative unit matrix. The purpose of the matrix is to ensure that after the cost functions have been maximised the next variables to be maximised in the lexicographic tableau are those in the original objective function.

Once the data has been read and the tableau assembled procedures Intsimp and Dintsimp are called to find the solution to the linear programming problem, and the result is printed. Procedure Integer is then called to find an integer solution, which is also printed.

A constraint is then added to make the solution just found infeasible. In the case of all but programme BGD this automatically terminates the programme. In the case of programme BGD procedure Integer is reentered to search for a better one.

m is initially the number of constraints in the original data but is later increased to include the extra constraints and cost functions generated by the programme,

n is the number of non-basic variables in the original data,

D is the absolute value of the determinant of the inverse matrix, and is initially set to 1,

R is used for numbering slack variables added in procedure Integer: it is initially equal to m+n,

L is used to define the dimensions of a and p,

x defines the dimension of integer array z,

s is the index of the row of a containing the objective function, and is thus also the number of added cost rows,

y is the number of constraints in the input data to be placed above the negative unit matrix. It is defined by one of the parameters in the data,

tm holds the run time at which timing was last started,

nores is <u>true</u> if no feasible integer solution has yet been found,

f, f8, g, h, i, j, k, and u are formats and working variables,

q initially holds the objective function: later it holds the indices identifying the non-basic variables,

p holds the indices identifying the basic variables,

a holds the tableau representing the problem to be solved,

z holds the parameters of the problem. Only one can be set by the original data and that is z[8] which holds y. If unspecified it is set to zero. The six elements of z which are not redundant are

z[1]=D, z[2]=R, z[3]=L,

z[7]= time used by the programme so far, excluding input/output,

z[8]=y, z[10]=s;

```
integer m,n,D,R,L,x,s,y,tm,f,f8,g,h,i,j,k,u;
boolean no res; nores:=true;
open(20); open(30);
START:  copytext(20,30,[↑↑]); m:=read(20); if m<0 then goto END;
if not nores then gap(30,1); nores:=true; n:=read(20);
x:=if n>0 then 0 else read(20); n:=abs(n);
begin integer array q[0:n],z[1:if x≥10 then x else 10];
comment The variable parameters are read in;
        for i:=1 step 1 until x,x+1 step 1 until 10 do
          z[i]:=if i>x then 0 else read(20);
        y:=z[8]; if x<10 then x:=10; g:=0;
comment The cost function is read into array q and the element with
  maximum absolute value assigned to g;
        for j:=0 step 1 until n do
          begin q[j]:=read(20); u:=inbasicsymbol(20);
                if u=37 or u=32 then
                  begin u:=if u=37 then 0 else 1; h:=q[j]-1;
                        for k:=0 step 1 until h do q[j+k]:=u;
                        j:=j+h
                  end;
                if j>0 and abs(q[j])>g then g:=abs(q[j]);
          end;
        s:=0;
comment In this position programme BGD calculates the number of additional
  cost rows to be added by including the following statement:
  for h:=1, h×2 while h≤g do s:=s+1;
        m:=m+n+s; L:=z[3]:=m+2; z[10]:=s;
```

```
begin integer array a[-1:L+1,0:n],p[0:L+1];
        switch FAIL:=F1,F2,F3,F4,F5,F6,F7,F8;
        procedure f1(s,ind); value ind; string s; boolean ind;
          begin newline(30,2);
                if ind then writetext(30,s)
                  else writetext(30,[no*better*solution]);
                goto FINISH
          end;
        q[0]:=-q[0];
comment The following statement generates the additional cost rows for
  programme BGD. In the other programmes s=0 and the cost function is
  simply copied from array q to row 0 of array a;
        for i:=0 step 1 until s do
          begin g:=2↑(s-i); p[i]:=2↑i;
                for j:=0 step 1 until n do a[i,j]:=(q[j]+g÷2×sign(q[j]))÷g;
          end;
        p[s]:=0;
        for j:=0 step 1 until n do
          begin a[-1,j]:=0; q[j]:=j end;
        R:=n;
comment The constraints on the input data are now read in and the
  artificial cost function generated;
```

```
for i:=s+1 step 1 until s+y,s+y+n+1 step 1 until m do
  begin R:=R+1; j:=read(20); g:=inbasicsymbol(20);
        p[i]:=if g=162 or g≠178 and g≠146 and j<0 then -R else R;
        g:=if g=178 then -1 else +1;
        for j:=1 step 1 until n do
          begin a[i,j]:=g×read(20);
                u:=inbasicsymbol(20);
                if u=37 or u=32 then
                  begin u:=if u=37 then 0 else g; h:=abs(a[i,j])-1;
                        for k:=0 step 1 until h do
                          begin a[i,j+k]:=u;
                                if p[i]<0
                                 then a[-1,j+k]:=a[-1,j+k]-a[i,j+k]
                          end;
                        j:=j+h
                  end
                else if p[i]<0 then a[-1,j]:=a[-1,j]-a[i,j]
          end;
        a[i,0]:=g×read(20); if p[i]<0 then a[-1,0]:=a[-1,0]-a[i,0]
  end;
z[1]:=D:=1; z[2]:=R; s:=z[10];
comment The negative n×n unit matrix is generated;
for i:=s+y+1 step 1 until s+y+n do
  begin for j:=0 step 1 until n do a[i,j]:=0;
        a[i,i-s-y]:=-1; p[i]:=i-s-y
  end;
```

```
        f:=format([s-ndddddsdddddd]); f8:=format([-nddddddd;]);
        tm:=time; z[7]:=0;
comment The feasible optimal solution in rationals is found and printed out;
        Intsimp(a,m,n,p,q,D,R,L,FAIL,x,z);
        Dintsimp(a,m,n,p,q,D,R,L,FAIL,x,z);
        z[7]:=time-tm; writetext(30,[[cc]rational*solution[15s]D=]);
        write(30,f,D); newline(30,2);
        for i:=0 step 1 until m do
          begin write(30,f,p[i]); write(30,f,a[i,0]); writetext(30,[*/]);
                write(30,f,D); space(30,10); output(30,a[i,0]/D)
          end; newline(30,2);
        tm:=time;
comment The integer solution to the problem is found and printed out;
REIT:   Integer(a,m,n,p,q,D,R,L,FAIL,x,z);
        z[7]:=z[7]+time-tm; writetext(30,[[cc]D=]); write(30,f8,D);
        writetext(30,[[10s]cost*is*variable*numbered]);
        write(30,f8,p[s]); newline(30,2);
        g:=m+6+1;
        for i:=0 step 1 until g-1 do
          for j:=i step g until i+5×g do
            if j<m then begin write(30,f8,p[j]); write(30,f8,a[j,0]) end
              else begin newline(30,1); j:=i+5×g end;
        writetext(30,[[cc]run*time*in*secs=]);
        write(30,f,z[7]); newline(30,2);
        if z[7]≥1800 then goto CLOSE; tm:=time; s:=z[10];
        if nores then m:=m+1; nores:=false;
```

```
comment A constraint is added to make the integer solution just found
  infeasible, so that a search can be made for a better one.  In all but
  programme BGD the first solution found is optimal so that the next
  five lines could be replaced by  goto FINISH;
        p[m]:=p[m+1]:=0; a[m,0]:=a[m+1,0]:=-D;
        for j:=1 step 1 until n do a[m,j]:=a[m+1,j]:=a[s,j];
        Intch(a,m+1,n,p,q,D,m+1,pivot(a,m,n,m,0,F4),FAIL,F6);
        Dintsimp(a,m,n,p,q,D,R,L,FAIL,x,z);
        goto REIT;
CLOSE:  printar(a,m,n,p,q,D,f);
        writetext(0,[[c]another*problem*completed]); goto START;
F8:     fl([array*full*up],true);
F7:     fl([lexicographically*unbounded],true);
F6:     fl([integer*overflow],true);
F5:     fl([logical*error],true);
F4:     fl([no*integer*solution],nores);
F3:     fl([no*rational*solution*(primal)],true);
F2:     fl([no*rational*solution*(dual)],nores);
F1:     fl([solution*unbounded],true);
FINISH: z[7]:=z[7]+time-tm; writetext(30,[[cc]run*time*in*secs=]);
        write(30,f,z[7]); newline(30,2); goto CLOSE;
end
end;
END:  close(20); close(30)
end→
```

APPENDIX D

PART 2

THE OTHER PROGRAMMES

Part 2 : The Other Programmes

The specification of the other programmes is the same as that of programme BHD. As they differ from BHD only within one or two procedures only the differences are reproduced here.

Programmes BHM, BH9, BHQ, BHN, BHP, BHE and BHF differ only in procedure Constraint. Their methods of choosing constraints are described in sections (b) to (h) of Part 4 of Chapter 2.

Programme BH6 uses the same version of procedure Constraint as does programme BHF but has its own version of integer procedure pivot. This is described in section (i) of Part 4 of Chapter 2.

```
procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BHM;
begin integer i,j,g;
        for i:=0 step 1 until m do if dlrem(a[i,0],1,D)≠0 then goto FND;
        finish:=true; goto DONE;
FND:    g:=1;
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;

procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BH9;
begin integer i,j,g;
        for i:=1 step 1 until m do if dlrem(a[i,0],1,D)≠0 then goto FND;
        finish:=true; goto DONE;
FND:    g:=(D-1)+dlrem(a[i,0],1,D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;
```

```
procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BHQ;
begin integer i,j,g;
        for i:=0 step 1 until m do if dlrem(a[i,0],1,D)≠0 then goto FND;
        finish:=true; goto DONE;
FND:    g:=D-euclidalg(a[i,0],D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;

procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BHN;
begin integer i,j,g,h,k;
        g:=0;
        for h:=0 step 1 until m do
          begin k:=dlrem(a[h,0],1,D);
                if k>g then begin i:=h; g:=k end
          end;
        if g=0 then begin finish:=true; goto DONE end;
        for j:=1 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],1,D);
        g:=euclidalg(a[i,pivot(a,m+1,n,m+1,0,fail)],D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;
```

```
procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BHP;
begin integer i,j,g;
        for i:=0 step 1 until m do if dlrem(a[i,0],1,D)≠0 then goto FND;
        finish:=true; goto DONE;
FND:    for j:=0 step 1 until n do
          a[m+1,j]:=if dlrem(a[i,j],1,D)=0 then 0 else -1;
        g:=euclidalg(a[i,pivot(a,m+1,n,m+1,0,fail)],D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;

procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that contained in programme BHE;
begin integer i,j,g,h,k;
        g:=0;
        for h:=0 step 1 until m do
          begin k:=dlrem(a[h,0],1,D);
                if k>g then begin i:=h; g:=k end
          end;
        if g=0 then begin finish:=true; goto DONE end;
        g:=1;
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;
```

```
procedure Constraint(a,m,n,p,D,fail,finish); value m,n,D;
integer array a,p; integer m,n,D; label fail; boolean finish;
comment The text of this procedure is that in programmes BHF and BH6;
begin integer i,j,g,h,k;
        g:=0;
        for h:=0 step 1 until m do
          begin k:=dlrem(a[h,0],1,D); if k≠0 then k:=(D-1)÷k×k;
                if k>g then begin i:=h; g:=k end
          end;
        if g=0 then begin finish:=true; goto DONE end;
        g:=(D-1)÷dlrem(a[i,0],1,D);
        for j:=0 step 1 until n do a[m+1,j]:=-dlrem(a[i,j],g,D);
        writetext(30,[C****]);
DONE:
end;
```

```
integer procedure pivot(a,m,n,I,i0,FAIL); value m,n,I,i0;
integer array a; integer m,n,I,i0; label FAIL;
comment This version of the procedure was used in programme BH6. It
  resolves ties between two possible pivot columns by choosing the
  first one, rather than referring to the following rows as in the
  lexicographic method. Nevertheless it still ensures that when
  called by Intsimp the artificial cost and objective functions are
  optimised simultaneously;
begin integer j,J,gn,gd;
        J:=gn:=0; gd:=-1;
        for j:=1 step 1 until n do if a[I,j]<0 and a[-1,j]≤0 then
          begin if dlsign(gn,a[I,j],a[i0,j],gd)<0 or J=0 then
                  begin gn:=a[i0,j]; gd:=a[I,j]; J:=j end
          end;
        if i0=-1 then
          begin for j:=1 step 1 until n do
                  if a[-1,j]=a[-1,J] and a[0,j]<a[0,J] then J:=j
          end;
        pivot:=J; if J=0 then goto FAIL
end;
```