

# Special Facilities in a General Mathematical Programming System for Non-convex Problems Using Ordered Sets of Variables

E. M. L. BEALE & J. A. TOMLIN

## INTRODUCTION

Branch and bound methods have proved successful in finding optimum or near-optimum solutions to a variety of non-convex problems. In particular, they have been applied to the solution of general mixed integer programming problems. The procedure described by Beale and Small [1], and in more detail by Beale [2], which is based on the earlier work of Land and Doig [4], Little, Murty, Sweeney and Karel [5], Dakin [6] and Driebeek [7], has been used successfully on many types of mixed integer programming problems.

Special branch and bound methods have also been developed for minimizing sums of non-convex functions of variables subject to linear constraints. An application of this approach is described by Beale [8], and the approach is discussed more generally by Lawler & Wood [9], by Falk & Soland [10], and by Rech & Barton [11]. On the other hand, as Markowitz and Manne [12] pointed out many years ago, such problems can be transformed into general mixed integer programming problems. It is therefore tempting to conclude that, having developed a good branch and bound code for general mixed integer programming problems, one does not need to provide further facilities for these non-convex problems. But an analysis of the branch and bound processes shows that the special procedure takes better advantage of the structure of these problems. We therefore considered how we might incorporate such a procedure into a general mathematical programming system. In general, special procedures are only worthwhile if:

- (a) they can be implemented reasonably easily
  - (b) they are reasonably easy to use
- and
- (c) they enable a significant class of problems to be solved more easily than they could by other means.

To achieve these objectives, the procedure must be based on a suitable unifying concept expressed at a technical level, that is to say a concept expressed in terms of the structure of the input data presented to the mathematical programming code, as opposed to a concept expressed in terms of the problem represented by these data.

For these non-convex functions, the appropriate concept is that of ordered sets of variables. Such ordered sets can be used in a branch and bound process if we introduce two markers, whose positions vary as we progress from one subproblem to another in the tree of alternative possibilities to be considered. The significance of the markers is that members of the set that do not lie between the markers are forced to take zero values. We say that these variables are 'flagged' out of the problem, the variables between the markers being the only 'unflagged' members of the set. The positions of the markers are then treated by the branch and bound process in the same way as the lower and upper bounds on general integer variables.

Once these concepts of ordered sets of variables and markers have been introduced, we can use them for a wider class of problems than those which originally motivated them. We have developed two facilities with them; and before considering the facility developed for non-convex functions it is convenient to consider one that is logically simpler from the point of view of the code.

The first facility is for sets in which not more than one member may take a non-zero value in the final solution. This is intended for problems characterized by Healy [3], as 'multiple choice programming problems'; for example on a site one may either do nothing, or else build a small plant, or else build a large plant—each possibility being represented by a single (integer) variable.

The second facility is for sets in which up to two members may take non-zero values, provided that they are consecutive. This is intended primarily for non-convex functions of single arguments. If such a function  $f(z)$  occurs in either the objective function or a constraint of the problem, we write  $\sum b_i \lambda_i$  for  $f(z)$  where

$$\sum a_i \lambda_i = z, \quad (1.1)$$

$$\sum \lambda_i = 1, \quad (1.2)$$

and the values of  $a_i$  and  $b_i$  are chosen so that  $f(z) = b_i$  when  $z = a_i$ , and the function is adequately approximated by linear interpolation between them. The variables  $\lambda_i$  must then all be non-negative, but in the final solution if as many as two are non-zero they must be adjacent. By branch and bound we can therefore find a global optimal solution to the problem for which separable

programming, as defined by Miller[13], gives a local optimum. As in separable programming, we identify (1.1) as a 'reference row', since this is the row to which to refer when calculating the effective argument  $z$  from the weights  $\lambda_i$ , and we identify (1.2) as the 'convexity row' for this set of variables. The convexity row may be treated as a generalized upper bound.

The detailed logic for the branching process required for each facility is discussed in the next section. The advantages of this approach over the conventional integer programming approach are then discussed. Finally, preliminary computational experience is indicated.

### BRANCH AND BOUND ALGORITHM

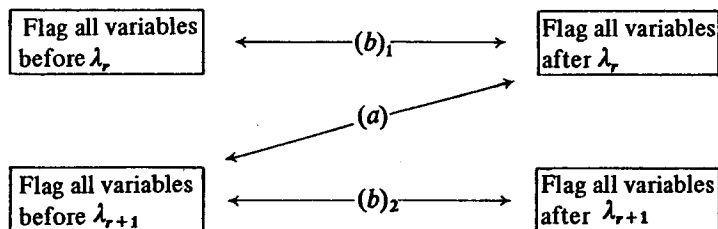
The basic principle to be exploited is that if  $\lambda_r$  and  $\lambda_{r+1}$  are any adjacent pair of variables of an ordered set then:

- Case (a) when only one variable of the set may take on a non-zero value *either* all variables beyond  $\lambda_r$  vanish *or* all variables before  $\lambda_{r+1}$  vanish.  
 Case (b) when two adjacent variables may be non-zero *either* all variables beyond  $\lambda_r$  vanish *or* all variables before  $\lambda_r$  vanish (and similarly for  $\lambda_{r+1}$ )

If at any stage of our branch and bound algorithm the current optimum trial solution contains unacceptable combinations of non-zero variables within the special ordered sets, we may branch to obtain new subproblems which exclude these combinations. This is done by flagging subsets of the variables in such a way as to bring the markers (between which variables are unflagged) closer together in the subproblems, in the same way as one varies the upper and lower bounds on integer variables for the mixed integer method.

Considering again any two adjacent variables  $\lambda_r, \lambda_{r+1}$  of an ordered set, the alternatives open to us for attempting to satisfy the conditions for cases (a) and (b) may be represented as in Figure 1.

Figure 1



It remains to choose a suitable pair of adjacent variables  $\lambda_r, \lambda_{r+1}$  and a systematic and efficient method of deciding which of the alternatives in Figure 1 to take.

A general procedure for finding a suitable pair of variables  $\lambda_r, \lambda_{r+1}$  for branching is to compute

$$\bar{w} = \sum_i w_i \lambda_i / \sum_i \lambda_i, \quad (2.1)$$

where the weights  $w_i$  are the reference row entries  $a_i$  if they exist and the sequence numbers  $i$  otherwise. The ends of the current interval, and hence the pair  $(\lambda_r, \lambda_{r+1})$  are determined by finding  $w_r$  and  $w_{r+1}$  such that

$$w_r \leq \bar{w} \leq w_{r+1}, \quad \text{or} \quad w_r \geq \bar{w} \geq w_{r+1} \quad (2.2)$$

When the appropriate pair of variables has been found, we must still choose which set to branch on and which branch to take first. In ordinary integer programming this is done by computing 'penalties' for each integer variable that takes a non-integer value in the current trial solution, representing lower limits on the costs of driving this variable either up or down to the nearest integer value. We then find the largest of these penalties and put the situation that gives such a penalty into the 'stored list' of subproblems to be considered later while we concentrate on the alternative branch for this variable. In our more general problem we must first consider how to calculate penalties for each of the possible courses of action indicated in Figure 1. We can then consider which penalties to actually calculate.

The tableau may be written in the form

$$\begin{aligned} x_0 &= \bar{a}_{00} + \sum \bar{a}_{0j}(-x_j) \\ X_i &= \bar{a}_{i0} + \sum \bar{a}_{ij}(-x_j), \end{aligned}$$

where  $x_0$  represents the objective function being maximized,  $X_i$  represents the  $i^{\text{th}}$  basic variable, and summation extends over all non-basic variables  $x_j$ . Then the penalty in normal integer programming associated with driving the basic variable  $X_i$  to zero would be

$$\bar{a}_{i0} \min_{j, \bar{a}_{ij} > 0} (\bar{a}_{0j} / \bar{a}_{ij}) \quad (2.3)$$

Let us now concentrate on case (a), and assume that no other ordered sets occur in our problem. We then calculate the penalties associated with the pair of alternatives (a) in Figure 1, and find the largest such penalty. If the problem contains general integer variables as well as these ordered sets, and if the largest penalty from an ordered set is smaller than the largest penalty from a general integer variable, then we branch on the general integer variable in the normal

way. But if the largest penalty is obtained from an ordered set, then the corresponding possible new positions of the markers are recorded on the 'stored list' of subproblems, while the alternative associated with the smaller penalty is explored.

In case (b) the situation is a little more complicated, since we do not know whether to consider the alternatives  $(b)_1$  or  $(b)_2$  in Figure 1. It is clear that the largest penalties for any of the four alternatives must be obtained by either flagging all variables after  $\lambda_r$  or flagging all variables before  $\lambda_{r+1}$ . We therefore normally calculate these penalties: if the former is the largest we take the alternatives  $(b)_1$ , and immediately explore the consequences of flagging all variables before  $\lambda_r$ ; otherwise we take the alternatives  $(b)_2$  and immediately explore the consequences of flagging all variables after  $\lambda_{r+1}$ . But, as in other applications of branch and bound, it is desirable to arrange that the branch explored immediately excludes the current trial solution. So before computing any penalties we see if any variable before  $\lambda_r$  is non-zero in the current trial solution and if not we exclude the alternatives  $(b)_1$ ; similarly, unless some variable after  $\lambda_{r+1}$  is non-zero in the current trial solution we exclude the alternatives  $(b)_2$ . (If no variable outside the range  $(\lambda_r, \lambda_{r+1})$  is non-zero, then there is nothing infeasible about this set and we can ignore it at this stage).

#### ADVANTAGES OF THE ALGORITHM

It is instructive to consider the advantages of this algorithm in general terms.

First consider Case (a), when just one of a number of variables must be selected. Typically this can be represented by a constraint of the form  $\sum \delta_j = 1$ , where the variables  $\delta_j$  have to be either zero or one. The conventional branch and bound integer programming approach is then to take a single variable from the set and force it to either zero or one. Forcing it to zero generally makes little progress towards a genuinely feasible solution. Forcing it to one automatically forces the others to zero and makes substantial progress. Beale and Small [1] therefore proposed special facilities for variables that are more suitable for forcing to one than to zero, such variables always being forced to one in the branch of the tree considered first. But practical experience with this facility has proved disappointing. In effect the decisions being made are too drastic: the code is committed to choosing a particular member of the set before there is enough evidence for a considered judgement. On the other hand, by dividing the variables into two subsets and choosing between the subsets the new facility approaches the selection problem in a more orderly way.

The point is particularly clear if the set has a reference row. Suppose for example that we are choosing whether to build a plant with capacity 0, 1, 4, 9 or

16 units and that the cost is proportional to the square root of the capacity. Then part of the problem may be formulated as follows:

$$\begin{aligned} C &= \delta_1 + 2\delta_2 + 3\delta_3 + 4\delta_4 \\ x - \delta_1 - 4\delta_2 - 9\delta_3 - 16\delta_4 &\leq 0 \\ \delta_0 + \delta_1 + \delta_2 + \delta_3 + \delta_4 &= 1, \end{aligned}$$

where  $C$  represents the cost of the plant, and  $x$  represents the used capacity. If the optimum value of  $x$  (determined by other factors not explicitly considered here) is about 3, then the linear programming solution will put  $\delta_0 = 13/16$  and  $\delta_4 = 3/16$ . If we now force either  $\delta_0$  or  $\delta_4$  to zero we make very limited progress towards a solution where the  $\delta_i$  are all integers. On the other hand if we force either  $\delta_0$  or  $\delta_4$  to one we are probably making an unfortunate choice. The new facility described in this paper will recognize  $(\delta_1, \delta_2)$  as the current interval, and will move one marker into this interval. If the upper marker is moved, then the next linear programming subproblem will be able to use only  $\delta_0$  and  $\delta_1$ , and may well choose  $\delta_1$  with weight 1; while if the lower marker is moved, then the next linear programming subproblem will be able to use only  $\delta_2, \delta_3$  and  $\delta_4$  and may well choose  $\delta_2$  with weight 1.

A similar situation arises if we change the problem to allow intermediate plant sizes. We now use the same ordered set of variables in Case (b). As Dantzig [14] pointed out, the problem can be formulated as a conventional mixed integer programming problem by defining new integer variables, say  $\lambda_1, \lambda_2, \lambda_3$  and  $\lambda_4$  representing the 'probability' that the solution lies in the first, second, third or fourth interval respectively, and adding 5 new constraints. Again, if 3 is a good value for  $x$ , the linear programming solution will put  $\delta_0 = 13/16$  and  $\delta_4 = 3/16$ , which involved putting  $\lambda_1 = 13/16$  and  $\lambda_4 = 3/16$ . If we now force either  $\lambda_1$  or  $\lambda_4$  to zero we make a very limited progress towards a solution where the  $\lambda_i$  are all integers. On the other hand if we force either  $\lambda_1$  or  $\lambda_4$  to one we are probably making an unfortunate choice. The new facility described in this paper will again recognize  $(\delta_1, \delta_2)$  as the current interval. If the largest penalty is associated with moving the upper marker into this interval, then the branch taken will be to move the lower marker into the interval  $(\delta_0, \delta_1)$  so that the next solution will probably involve  $\delta_1$  and  $\delta_4$ . This is a useful move, since the next branch on this set will probably move the upper marker into the interval  $(\delta_1, \delta_2)$  and give a valid combination of variables in this set. Similar progress is made if the largest penalty is associated with moving the lower marker into the interval  $(\delta_1, \delta_2)$ .

## PRELIMINARY COMPUTATIONAL EXPERIENCE

The facilities described in this paper have been incorporated in UMPIRE, a Universal Mathematical Programming system Incorporating Refinements and Extensions written by Scientific Control Systems (Scicon) for the Univac 1108 computer.

The very limited computational experience available at the time of writing supports the theoretical arguments of the previous Section. But the only comparative statistics worth quoting are for a small test problem involving the minimization of the sum of two concave functions of single arguments. The new facility produced the optimal solution and completed the search in only 6 nodes and 39 simplex iterations, while the conventional mixed integer approach involved 20 nodes and 227 simplex iterations.

## REFERENCES

1. E. M. L. BEALE and R. E. SMALL (1965) Mixed integer programming by a branch and bound technique. *Proceedings of the IFIP Congress 1965* 2 450-1. Edited by W. A. Kalenich, Spartan Press, Washington D.C. and Macmillan, London.
2. E. M. L. BEALE (1968) *Mathematical Programming in Practice*. Pitmans, London.
3. W. C. HEALY JR. (1964) Multiple choice programming. *Opns Res.* 12 122-38.
4. A. H. LAND and A. G. DOIG (1960) An automatic method for solving discrete programming problems. *Econometrica* 28 497-520.
5. J. D. C. LITTLE, K. C. MURTY, D. W. SWEENEY and C. KAREL (1963) An algorithm for the travelling salesman problem. *Opns Res.* 11 972-89.
6. R. J. DAKIN (1965) A tree-search algorithm for mixed integer programming problems. *The Computer Journal* 8 250-5.
7. N. J. DRIEBEEK (1966) An algorithm for the solution of mixed integer programming problems. *Mgmt Sci.* 12 576-87.
8. E. M. L. BEALE (1964) Two transportation problems. *Proceedings of the Third IFORS Conference, Oslo 1963*, 780-8. Edited by G. Kreweras and G. Morlat. Dunod Paris and E.U.P. London.
9. E. L. LAWLER and D. E. WOOD (1966) Branch and bound methods. A survey. *Opns Res.* 14 699-719.

10. J. E. FALK and R. M. SOLAND (1968) An algorithm for separable non-convex programming problems. Research Analysis Corporation, McLean, Virginia, U.S.A.
11. P. RECH and L. G. BARTON (1969) A non-convex transportation algorithm. *Proceedings of the NATO Conference on Applications of Mathematical Programming*, Cambridge 1968. Edited by E. M. L. Beale. E.U.P. London.
12. H. M. MARKOWITZ and A. S. MANNE (1957) On the solution of discrete programming problems. *Econometrica* 25 84-110.
13. C. E. MILLER (1963) The simplex method for local separable programming. *Recent Advances in Mathematical Programming* 89-100. Edited by R. L. Graves and P. Wolfe. McGraw Hill, New York.
14. G. B. DANTZIG (1960) On the significance of solving linear programming problems with some integer variables. *Econometrica* 28 30-44.