Chapter h – Operations Research

1. Scope of the Chapter

This chapter contains functions to solve certain integer programming problems, and the transportation problem. The functions provided in this chapter treat all matrices as dense. They are therefore suitable for reasonable sized problems, but are less suited to large sparse problems for which specialist software might be more appropriate. For large sparse LP or QP problems, use nag_opt_sparse_convex_qp (e04nkc).

2. Background

General *linear programming* (LP) and *quadratic programming* (QP) problems (see Dantzig (1963)) are of the form:

$$\text{minimize} \quad F(x), x \in R^n$$

where $F(x)$ is of the form $c^T x$ (LP) or $c^T x + \frac{1}{2} x^T H x$ (QP) for some constant vector $c \in R^n$, matrix $H \in R^{n \times n}$, subject to linear constraints which may have the forms:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad i = 1, 2, \ldots, m_1 \qquad \text{(equality)}$$

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = m_1 + 1, \ldots, m_2 \qquad \text{(inequality)}$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad i = m_2 + 1, \ldots, m \qquad \text{(inequality)}$$

$$x_j \geq l_j \qquad j = 1, 2, \ldots, n \qquad \text{(simple bound)}$$

$$x_j \leq u_j \qquad j = 1, 2, \ldots, n \qquad \text{(simple bound)}$$

This chapter deals with *integer programming* (IP) problems in which some or all the elements of the solution vector x are further constrained to be *integers*. For general LP and QP problems where x takes only real (i.e., non-integer) values, refer to Chapter e04.

IP problems may or may not have a solution, which may or may not be unique.

Consider for example the following problem:

$$\begin{array}{lrcl} \text{minimize} & 3x_1 & + & 2x_2 \\ \text{subject to} & 4x_1 & + & 2x_2 \geq 5 \\ & & & 2x_2 \leq 5 \\ & x_1 & - & x_2 \leq 2 \\ \text{and} & x_1 & \geq & 0, x_2 \geq 0. \end{array}$$

The shaded region in Figure 1 is the *feasible region*, the region where all the constraints are satisfied, and the points within it which have integer co-ordinates are circled. The lines of shading are in fact contours of decreasing values of the objective function $3x_1 + 2x_2$, and it is clear from Figure 1 that the optimum IP solution is at the point (1,1). For this problem the solution is unique.

However, there are other possible situations:

(a) there may be more than one solution; e.g., if the objective function in the above problem were changed to $x_1 + x_2$, both (1,1) and (2,0) would be IP solutions.

(b) the feasible region may contain no points with integer co-ordinates, e.g., if an additional constraint

$$3x_1 \leq 2$$

were added to the above problem.

(c) there may be no feasible region, e.g., if an additional constraint

$$x_1 + x_2 \leq 1$$

were added to the above problem.

(d) the objective function may have no finite minimum within the feasible region; this means that the feasible region is unbounded in the direction of decreasing values of the objective function, e.g., if the constraints

$$4x_1 + 2x_2 \geq 5, \quad x_1 \geq 0, \quad x_2 \geq 0,$$

were deleted from the above problem.

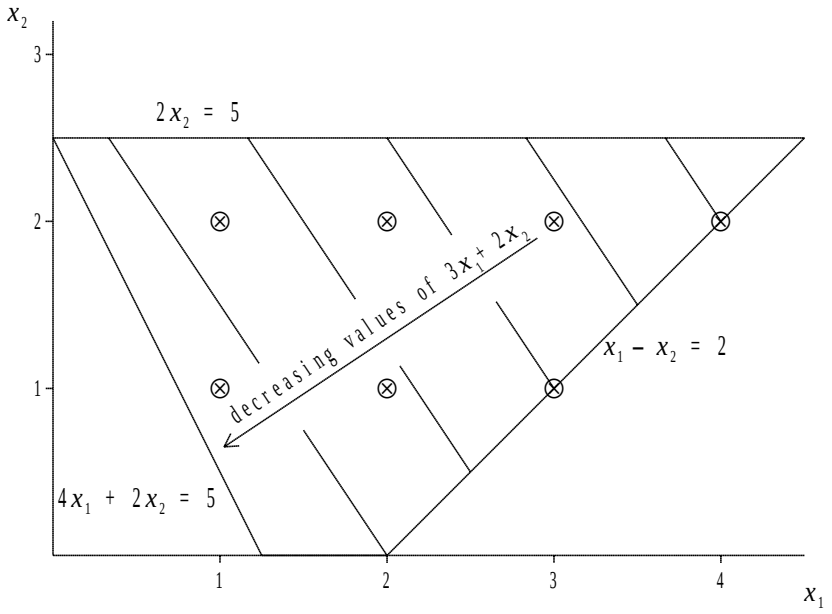

Figure 1

Algorithms for IP problems are usually based on algorithms for general LP or QP problems, together with some procedure for constructing additional constraints which exclude non-integer solutions (see Beale (1977)). The basic principle is the same whether the objective is of the LP or QP form.

The Branch and Bound (BB) method is a well-known and widely used technique for solving IP problems (see Beale (1977) or Mitra (1973)). It involves subdividing the optimum solution to the original LP (or QP) problem into two mutually exclusive sub-problems by branching an integer variable that currently has a fractional optimal value. Each sub-problem can now be solved as an LP (or QP) problem, using the objective function of the original problem. The process of branching continues until a solution for one of the sub-problems is feasible with respect to the integer problem. In order to prove the optimality of this solution, the rest of the sub-problems in the BB tree must also be solved. Naturally, if a better integer feasible solution is found for any sub-problem, it should replace the one at hand.

A common method for specifying IP and LP/QP problems in general is the use of the MPSX file format (see unattributed (1971)). A full description of this file format is provided in the routine documents for nag_ip_mps_read (h02buc) and nag_opt_sparse_mps_read (e04mzc).

The efficiency in computations is enhanced by discarding inferior sub-problems. These are problems in the BB search tree whose LP (or QP) solutions are greater than (in the case of minimization) the best integer solution at hand.

A special type of linear programming problem is the *transportation* problem in which there are $p \times q$ variables y_{kl} which represent quantities of goods to be transported from each of p sources to each of q destinations.

The problem is to minimize

$$\sum_{k=1}^{p}\sum_{i=1}^{q} c_{ki}y_{ki}$$

where c_{kl} is the unit cost of transporting from source k to destination l. The constraints are:

$$\sum_{i=1}^{q} y_{ki} = A_k \qquad \text{(availabilities)}$$

$$\sum_{k=1}^{p} y_{ki} = B_i \qquad \text{(requirements)}$$

Note that the availabilities must equal the requirements:

$$\sum_{k=1}^{p} A_k = \sum_{i=1}^{q} B_i = \sum_{k=1}^{p}\sum_{i=1}^{q} y_{ki}.$$

and if all the A_k and B_l are integers, then so are the optimal y_{kl}.

3. Recommendation on Choice of Function

This chapter contains functions to solve certain integer programming problems, and the transportation problem. The functions provided in this chapter treat all matrices as dense. They are therefore suitable for reasonable sized problems, but are less suited to large sparse problems for which specialist software might be more appropriate. For large sparse general LP or QP problems, use nag_opt_sparse_convex_qp (e04nkc).

4. Optional Facilities

The IP function in Chapter h provides a range of optional facilities: these offer the possibility of fine control over many of the algorithmic parameters and the means of adjusting the level and nature of the printed results. The MPSX reading function also offers some optional facilities.

Control of these optional facilities is exercised by a structure of type Nag_H02_Opt, the members of the structure being optional input or output parameters to the function. After declaring the structure variable, which is named **options** in this manual, the user must initialise the structure by passing its address in a call to the utility function nag_ip_init (h02xxc). Selected members of the structure may then be set to the user's required values and the address of the structure passed to the NAG function. Any member which has not been set by the user will indicate to the function that the default value should be used for this parameter. A more detailed description of this process is given below in Section 4.4.

Examples of parameters which may be altered from their default value are **options.feas_tol** and **options.int_tol** (these control the accuracy to which the constraints are satisfied in the BB sub-problems and the accuracy of the final objective function value, respectively), and **options.max_iter** (which limits the number of iterations the algorithm will perform at each sub-problem). Certain members of **options** supply further details concerning the final results, for example on exit from the IP solver the member pointers **options.state** and **options.lambda** give the status of the constraints and the final values of the Lagrange multipliers respectively. Another use of the **options** structure is to allow additional information read in by the MPSX reader (such as the MPSX row and column names) to be communicated to the IP solver for use in its printout.

4.1. Control of Printed Output

Results from the IP solution process are printed by default on the `stdout` (standard output) stream. These include the results after each node of the BB search tree and the final results at termination of the search process. The amount of detail printed out may be increased or decreased by setting the optional parameter **print_level**, i.e., the structure member **options.print_level**. This member is an `enum` type, Nag_PrintType, and an example value is **Nag_Soln** which when assigned to

options.print_level will cause the IP function to print only the final result; all intermediate results printout is suppressed.

If the results printout is not in the desired form then it may be switched off, by setting **options.print_level** = **Nag_NoPrint**, or alternatively the user can supply his or her own function to printout or make use of both the intermediate and final results. Such a function would be assigned to the pointer to function member **options.print_fun**; the user defined function would then be called in preference to the NAG print function.

In addition to the results, the values of the parameters to the optimization function are printed out when the function is entered; the Boolean member **options.list** may be set to **FALSE** if this listing is not required.

Printing may be output to a named file rather than to `stdout` by providing the name of the file in the **options** character array member **outfile**. Error messages will still appear on `stderr`, if **fail.print** = **TRUE** or the **fail** parameter is not supplied (see the Introduction to the NAG C Library Manual for details of error handling within the library).

The level of output provided by the MPSX reading function may also be controlled. In this case, control is provided by the optional parameter **output_level**.

4.2. Memory Management

The **options** structure contains a number of pointers for the input of data and the output of results. The NAG functions will manage the allocation of memory to these pointers; when all calls to these functions have been completed then a utility function nag_ip_free (h02xzc) can be called by the user's program to free the NAG allocated memory which is no longer required.

If the calling function is part of a larger program then this utility function allows the user to conserve memory by freeing the NAG allocated memory before the **options** structure goes out of scope. nag_ip_free (h02xzc) can free all NAG allocated memory in a single call, but it may also be used selectively. In this case the memory assigned to certain pointers may be freed leaving the remaining memory still available; pointers to this memory and the results it contains may then be passed to other functions in the user's program without passing the structure and all its associated memory.

Although the NAG C Library functions will manage all memory allocation and deallocation, it may occasionally be necessary for the user to allocate memory to the **options** structure from within the calling program before entering the optimization function.

An example of this is where the user stores information in a file from an optimization run and at a later date wishes to use that information to solve a similar optimization problem or the same one under slightly changed conditions. The pointer **options.state**, for example, would need to be allocated memory by the user before the status of the constraints could be assigned from the values in the file.

If the user does assign memory to a pointer within the **options** structure then the deallocation of this memory must also be performed by the user; the utility function nag_ip_free (h02xzc) will only free memory allocated by NAG C Library optimization functions. When user allocated memory is freed using the standard C library function `free()` then the pointer should be set to NULL immediately afterwards; this will avoid possible confusion in the NAG memory management system if a NAG function is subsequently entered.

4.3. Reading Optional Parameter Values From a File

Optional parameter values may be placed in a file by the user and the function nag_ip_read (h02xyc) used to read the file and assign the values to the **options** structure. This utility function permits optional parameter values to be supplied in any order and altered without recompilation of the program. The values read are also checked before assignment to ensure they are in the correct range for the specified option. Pointers within the **options** structure cannot be assigned to using nag_ip_read (h02xyc).

4.4. Method of Setting Optional Parameters

The method of using and setting the optional parameters is:

1 Declare a structure of type Nag_H02_Opt.

2 Initialise the structure using nag_opt_init (e04xxc).

3 Assign values to the structure.

4 Pass the address of the structure to the optimization function.

5 Call nag_opt_free (e04xzc) to free any memory allocated by the optimization function.

If after step 4, it is wished to re-enter the optimization function, then step 3 can be returned to directly, i.e., step 5 need only be executed when all calls to the optimization function have been made.

At step 3, values can be assigned directly and/or by means of the option file reading function nag_ip_read (h02xyc). If values are only assigned from the options file then step 2 need not be performed as nag_ip_read (h02xyc) will automatically call nag_ip_init (h02xxc) if the structure has not been initialised.

5. References

Ahuja R K, Magnanti T L and Orlin J B (1993) *Network Flows: Theory, Algorithms, and Applications* Prentice Hall.

Beale E M (1977) Integer Programming *The State of the Art in Numerical Analysis* (ed D A H Jacobs) Academic Press.

Dantzig G B (1963) *Linear Programming and Extensions* Princeton University Press.

Mitra G (1973) Investigation of branch and bound strategies for the solution of mixed integer linear programs *Math. Programming* **4** 155–170.

Williams H P (1990) *Model Building in Mathematical Programming* (3rd Edition) Wiley.

unattributed (1971) Program Number 5734 XM4 *MPSX - Mathematical programming system* IBM Trade Corporation, New York.

6. Available Functions

Solution of IP problems (linear or quadratic objective function)	`h02bbc`
IP utility functions	
read MPSX data for IP, LP or QP problem from a file	`h02buc`
free memory allocated by nag_ip_mps_read (h02buc)	`h02bvc`
initialize option structure to null values	`h02xxc`
read optional parameter values from a file	`h02xyc`
free NAG allocated memory from option structures	`h02xzc`
Transportation problem	`h03abc`