A C CODE FOR SOLVING THE GENERALIZED ASSIGNMENT PROBLEM

Nejat Karabakal
Department of Industrial & Operations Engineering
University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 92-32
May 1992

`kblgap`
A C Code for Solving the Generalized Assignment Problem*

Nejat Karabakal
Department of Industrial and Operations Engineering
The University of Michigan, Ann Arbor, Michigan 48109

Version 1.0 of May 1992

Contents

*This research was supported by National Science Foundation grants PYI-G-DMC-8352346 and DMM-9018515 to The University of Michigan.

1 Introduction

This report is a documentation and user manual for the C code, **kblgap**, developed for solving the following formulation of the generalized assignment problem (GAP):

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}\, x_{ij} & \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij}\, x_{ij} \le b_i & i = 1,\ldots,m \\ & \sum_{i=1}^{m} x_{ij} = 1 & j = 1,\ldots,n \\ & x_{ij} \in \{0,1\} & \text{all } i,j \end{aligned}$$

The code implements a branch-and-bound algorithm featuring an efficient Lagrangian relaxation methodology for bounding. Computational tests show that **kblgap** is, on the average, 100 times faster than the best branch-and-bound code (Martello-Toth code) available for the GAP to date. In addition, it is capable of solving larger problems (up to 500 variables) optimally in reasonable times. For more information, see Karabakal, Bean, and Lohmann (1992). Also, see Martello and Toth (1990) for computational comparisions of the Martello and Toth algorithm with other existing branch-and-bound algorithms, making it possible to compare **kblgap** with many other codes indirectly.

2 User Manual

2.1 Compiling and Running

The code is written with portability considerations in mind. The author has successfully compiled it under MS-DOS using Turbo C compiler,

```
tcc kblgap.c
```

and under Unix,

```
cc kblgap.c -okblgap -O
```

to produce the executable program. The executable kblgap is invoked by the command

kblgap ⟨*input-file-name*⟩

If ⟨*input-file-name*⟩ is omitted, kblgap prompts the user to enter an input file name. The output goes to the standard output (default is the screen). If a file output is desired, output redirection can be used (under Unix and MS-DOS), e.g.,

kblgap ⟨*input-file-name*⟩ > output.txt

The input file format is given below. All numbers are integers and seperated by at least one space.

$$\begin{array}{cccc}
m & n & & \\
c_{11} & c_{12} & \dots & c_{1n} \\
a_{11} & a_{12} & \dots & a_{1n} \\
c_{21} & c_{22} & \dots & c_{2n} \\
a_{21} & a_{22} & \dots & a_{2n} \\
\vdots & \vdots & \vdots & \vdots \\
c_{m1} & c_{m2} & \dots & c_{mn} \\
a_{m1} & a_{m2} & \dots & a_{mn} \\
b_1 & b_2 & \dots & b_m
\end{array}$$

2.2 Setting Parameters

The maximum problem size is given by three parameters that are used for dimensioning arrays.

`MAXM`	maximum number of knapsack constraints,
`MAXN`	maximum number of multiple-choice constraints,
`MAXB`	largest knapsack RHS capacity.

They are specified at the beginning with `#define` statements (see section 3.1). If these limits are violated, `kblgap` terminates with a descriptive error message. These three parameters are "static" in the sense that whenever any of them are modified, the code must be recompiled.

In contrast, the following parameters, used for fine-tuning the algorithmic performance and for controlling the detail of output, are "dynamic" in the sense that they can be specified in a seperate configuration file called `kblgap.cfg`.

`INFINITY`	a big number,
`ROOTMAMITLIM`	maximum multiplier adjustment method (MAM) iterations made at the root node of the branch-and-bound tree,
`MAMITLIM`	maximum MAM iterations made at any other node,
`ROOTSUBITLIM`	maximum subgradient iterations made at the root node,
`SUBITLIM`	maximum subgradient iterations allowed at any other node,
`SUBPATIENCE`	number of iterations the subgradient algorithm permits the upper bound to diverge before halving the scale parameter u (see `subgradient()` function description for more information),
`MAXBRANCH`	maximum number of nodes that can be generated in the branch-and-bound tree,
`PRINTSTEPS`	set to 1 if branching steps are to be printed, 0 otherwise,
`PRINTINPUT`	set to 1 if input data are to be printed, 0 otherwise,
`PRINTOPTIM`	set to 1 if the final optimum solution is to be printed, 0 otherwise,
`ZEROTOL`	zero tolerance.

Each time `kblgap` is run, it first checks the existence of `kblgap.cfg`. If it exists, values of the above parameters are read in from this file; otherwise, the default values assigned at the beginning of the code take effect (see section 3.1). Each line of the configuration file consists of two quantities, the value of the parameter followed by a descriptive string. Parameters must be supplied in the above order. A sample `kblgap.cfg` file is provided below.

```
32000    INFINITY
10       ROOTMAMITLIM
5        MAMITLIM
25       ROOTSUBITLIM
10       SUBITLIM
4        SUBPATIENCE
5000     MAXBRANCH
0        PRINTSTEPS
1        PRINTINPUT
1        PRINTOPTIM
0.001    ZEROTOL
```

Iteration limits for upper bounding procedures, **MAM()** and **subgradient()**, can be specified zero to prevent invoking a particular procedure.

The parameter, **MAXBRANCH**, controls the accuracy of the solution. If an optimum solution is sought, a big integer is assigned to **MAXBRANCH**. On the other hand, a fairly good feasible solution with an error bound can be obtained by keeping this parameter low. In the latter case, the tightest upper bound available at the time of termination is used for computing the maximum error of the best feasible solution. A line of output with the following format is printed:

```
Best solution found = (1), Upper bound = (2) (Error bound = (3)%)
```

where

1. Objective value of the best feasible solution,
2. Tightest upper bound on optimum objective,
3. $\frac{(2)-(1)}{(1)} \times 100$.

See section 2.3 for an example.

If the parameter PRINTSTEPS is turned on (set to one), a line with the following format is printed *for every node* generated in the branch-and-bound tree.

```
Node (1) From (2):  (3) [(4)] ==> [(5)] (6)
```

where

1. node's serial number,
2. parent node,
3. variable fixed at this node,
4. [current incumbent, upper bound of the subproblem at the parent node] pair,
5. [*potentially* better incumbent, upper bound of the subproblem at the current node] pair,
6. whether the node is fathomed or dangles after bounding.

Node 0 corresponds to the root node and its output line has a slightly different interpretation.

```
Node 0 :   [(1)] ==> [(2)] (3)
```

where

1. the interval [abslb, absub], where

   $$\texttt{abslb} = \sum_j \min\{c_{ij}\} \qquad \text{and} \qquad \texttt{absub} = \sum_j \max\{c_{ij}\}$$

   These quantities represent absolute lower and upper bounds, respectively, on the optimum objective of the GAP.

2. [best feasible solution, upper bound] pair for the original problem,

3. whether the root is fathomed or dangles after bounding (if it is fathomed, the best feasible solution is optimal; if it dangles, there is a duality gap).

If the number of nodes generated is expected to be high, PRINTSTEPS switch should be turned off to prevent excessive output. See section 2.3 for examples.

2.3 An Example

Consider a GAP with 5 knapsack and 10 multiple-choice constraints. Given below are an input data file 5by10.c1 set up for this problem and the code's output.

$$(c_{ij}) = \begin{pmatrix} 19 & 14 & 21 & 19 & 10 & 24 & 13 & 11 & 22 & 16 \\ 23 & 22 & 18 & 21 & 19 & 12 & 13 & 10 & 18 & 19 \\ 18 & 19 & 21 & 25 & 17 & 10 & 22 & 14 & 25 & 22 \\ 21 & 23 & 23 & 25 & 10 & 22 & 16 & 13 & 19 & 12 \\ 12 & 23 & 21 & 17 & 22 & 22 & 18 & 11 & 24 & 24 \end{pmatrix}$$

$$(a_{ij}) = \begin{pmatrix} 24 & 18 & 5 & 8 & 24 & 23 & 7 & 24 & 15 & 18 \\ 5 & 18 & 11 & 13 & 22 & 20 & 14 & 19 & 21 & 9 \\ 15 & 11 & 15 & 14 & 24 & 18 & 12 & 12 & 8 & 21 \\ 22 & 24 & 19 & 12 & 13 & 22 & 15 & 16 & 12 & 23 \\ 19 & 5 & 20 & 9 & 11 & 22 & 14 & 18 & 5 & 16 \end{pmatrix} \qquad (b_i) = \begin{pmatrix} 26 \\ 24 \\ 24 \\ 28 \\ 22 \end{pmatrix}$$

```
  5  10
 19    14    21    19    10    24    13    11    22    16
 24    18     5     8    24    23     7    24    15    18
 23    22    18    21    19    12    13    10    18    19
  5    18    11    13    22    20    14    19    21     9
 18    19    21    25    17    10    22    14    25    22
 15    11    15    14    24    18    12    12     8    21
 21    23    23    25    10    22    16    13    19    12
 22    24    19    12    13    22    15    16    12    23
 12    23    21    17    22    22    18    11    24    24
 19     5    20     9    11    22    14    18     5    16
26  24  24  28  22
```

```
---PARAMETERS SET-----------------------------------------------------
                    MAM   SUB     PRINT FLAGS       GENERAL
  Root iter limit     5    20     Input      YES    Infinity    32000
  Node iter limit     5    10     Branches   YES    Zero Tol    0.0010
  Patience            4           OptimSol   YES    Max Branch  10000
----------------------------------------------------------------------
  Program Limits: Max Rows 5, Max Columns 20, Max Knapsack RHS 200.
----------------------------------------------------------------------
PROBLEM DATA: 5 rows by 10 columns  (Input file: 5by10.c1)

Objective coefficients:
 19 14 21 19 10 24 13 11 22 16
 23 22 18 21 19 12 13 10 18 19
 18 19 21 25 17 10 22 14 25 22
 21 23 23 25 10 22 16 13 19 12
 12 23 21 17 22 22 18 11 24 24

Knapsack constraints:
 24 18  5  8 24 23  7 24 15 18 <= 26
  5 18 11 13 22 20 14 19 21  9 <= 24
 15 11 15 14 24 18 12 12  8 21 <= 24
 22 24 19 12 13 22 15 16 12 23 <= 28
 19  5 20  9 11 22 14 18  5 16 <= 22

Node 0 : [134, 225] ==> [201,215]  DANGLING
Node 1 From 0: x(1,9) [201,215] ==> [201,204]  DANGLING
Node 2 From 0: x(2,9) [201,215] ==> [201,200]  FATHOM
Node 3 From 0: x(3,9) [201,215] ==> [201,208]  DANGLING
Node 4 From 0: x(4,9) [201,215] ==> [201,205]  DANGLING
Node 5 From 0: x(5,9) [201,215] ==> [209,212]  DANGLING
Node 6 From 5: x(1,2) [209,212] ==> [209,204]  FATHOM
Node 7 From 5: x(2,2) [209,212] ==> [209,209]  FATHOM
Node 8 From 5: x(3,2) [209,212] ==> [209,208]  FATHOM
Node 9 From 5: x(4,2) [209,212] ==> [209,206]  FATHOM
Node 10 From 5: x(5,2) [209,212] ==> [209,209]  FATHOM

Optimum = 209
No of branches = 10

 cx = 209
   1234567890  USED  RHS
 1)..11......    13   26
 2)1........1    14   24
 3)......11..    24   24
 4).....1....    22   28
 5).1..1...1.    21   22
```

To illustrate the pieces of information printed for each branch-and-bound node, consider node 9. It is generated by branching from node 5 by fixing x_{42}. Hence, the subproblem at node nine has two fixed variables, x_{42} and x_{59}. Upper bounding procedures applied to this subproblem conclude that its optimum cannot exceed 206. Therefore, node 9 is fathomed.

If the parameter **MAXBRANCH** is set to 0, the output would look like the following:

```
---PARAMETERS SET-----------------------------------------------------
                    MAM   SUB     PRINT FLAGS        GENERAL
  Root iter limit    10    25     Input      NO      Infinity     32000
  Node iter limit     5    10     Branches   YES     Zero Tol     0.0010
  Patience            4           OptimSol   YES     Max Branch   0
----------------------------------------------------------------------
  Program Limits: Max Rows 5, Max Columns 20, Max Knapsack RHS 200.
----------------------------------------------------------------------
PROBLEM DATA: 5 rows by 10 columns  (Input file: 5by10.c1)

Node 0 : [134, 225] ==> [201,215]  DANGLING
Branch limit exceeded.

Best solution found = 201,  Upper bound = 215  (Error bound = 6.97%)
No of branches = 0

 cx = 201
   1234567890  USED  RHS
 1)..11..1...    20   26
 2)1........1    14   24
 3).......11.    20   24
 4).....1....    22   28
 5).1..1.....    16   22
```

3 Code Documentation

3.1 Basic Definitions and Data Structure

```
#include <stdio.h>

 /* Maximum problem size, modify to solve bigger problems */
#define MAXM 5      /* max no. of knapsack constraints */
#define MAXN 20     /* max no. of multiple-choice constraints */
#define MAXB 200    /* biggest knapsack capacity */

 /* Mnemonic definitions */
#define YES 1
#define NO  0
typedef int bool;

 /* Structure of a branch-and-bound tree node */
typedef struct NODE {
          int nodeno,          /* node's serial no */
              bound,           /* upper bound of the node's partial problem */
              ifix, jfix,      /* indices of the fixed variable */
              lambda[MAXN+1];  /* multipliers used in upper bounding */
          struct NODE *next, *parent;    /* pointers (see below) */
        } NODE, *NODEPTR;
```

The branch-and-bound tree consists of nodes as defined above. Each outstanding node is a member of either or both of the two linked lists maintained for efficient operations:

1. *Path from node to the root*: Each node points to its parent through the `*parent` link, except for the root node. The root's `*parent` is set to `NULL` indicating the end of a list traversal. By following the path from a particular node to the root, the algorithm determines which variables are fixed in the partial problem associated with this node. Another use of the `*parent` pointer is to initialize the multiplier vector with the best multipliers computed at the parent node.

2. *Dangling nodes*: This is a priority queue consisting only of dangling nodes, i.e., those without children. It is ordered by decreasing `bound` values. Nodes are connected via `*next` links. The start of the queue is pointed by `firstdangling`, so that `firstdangling->bound` always gives the highest upper bound among all yet-to-be-solved subproblems. When `firstdangling = NULL`, the branch-and-bound algorithm is terminated because this condition implies there exists no upper bound greater than the incumbent (see `BAB()` function for more information).

```
 /* Set defaults for parameters */
 /* (They can be modified by the use of configuration file kblgap.cfg) */
```

```
int   INFINITY     = 32000,
      ROOTMAMITLIM = 5,
      MAMITLIM     = 5,
      ROOTSUBITLIM = 20,
      SUBITLIM     = 10,
      SUBPATIENCE  = 4,
      MAXBRANCH    = 10000,
      PRINTSTEPS   = YES,
      PRINTINPUT   = YES,
      PRINTOPTIM   = YES;
float ZEROTOL = 0.001;
```

/* Global variables */

```
int m, n,                   /* actual no of knapsack and MC constraints */
    c[MAXM+1][MAXN+1],      /* objective values */
    a[MAXM+1][MAXN+1],      /* knapsack weights */
    b[MAXM+1],              /* knapsack capacities, or RHS values */
    abslb,                  /* sum of min objective values over every MC set */
    nbranch = 0,            /* branch counter */
    incumbent, xinc[MAXN+1]; /* incumbent value and solution */
```

Feasible solutions handled in various routines are kept in n-dimensional vectors, called *positions.* Let x is a feasible solution to the GAP and pos is the corresponding position vector. Then, for each j, pos[j] = i iff $x_{ij} = 1$. Incumbent solution xinc is a position vector.

```
NODEPTR firstdangling = NULL;   /* initially, there is no dangling nodes */
```

3.2 Input Functions

```
void readin(inpname)
char inpname[];
{
   FILE *inp;
   int i, j;

   if (inpname[0] == '*') {
      printf("Enter input file : ");  scanf("%s", inpname);
   }
   if ((inp = fopen(inpname, "r")) == NULL) {
      printf( "ERROR: Cannot open input file: %s\n", inpname);
      exit(1);
   }
```

```
fscanf(inp, "%d%d\n", &m, &n);
if (m > MAXM) {
   printf("ERROR: Row limit exceeded. ");
   printf(" (Data = %d, Limit %d)\n", m, MAXM);
   exit(1);
}
if (n > MAXN) {
   printf("ERROR: Column limit exceeded. ");
   printf(" (Data = %d, Limit %d)\n", n, MAXN);
   exit(1);
}

for (i=1; i <= m; i++) {
   for (j=1; j<=n; j++) {
      fscanf(inp, "%d", &c[i][j]);
      if (c[i][j] < 0) {
         printf("ERROR: Nonnegative objective value expected.");
         printf("  (Data: c[%d,%d] = %d)\n", i, j, c[i][j]);
         exit(1);
      }
   }
   fscanf(inp, "\n");
   for (j=1; j<=n; j++) {
      fscanf(inp, "%d", &a[i][j]);
      if (a[i][j] <= 0) {
         printf("ERROR: Positive knapsack weight expected.");
         printf("  (Data: a[%d,%d] = %d)\n", i, j, a[i][j]);
         exit(1);
      }
   }
   fscanf(inp, "\n");
}

for (i=1; i <= m; i++) {
   fscanf(inp, "%d", &b[i]);
   if (b[i] > MAXB) {
      printf("ERROR: Knapsack RHS too big.");
      printf("  (Data: b[%d] = %d, Limit %d)\n", i, b[i], MAXB);
      exit(1);
   }
}
fscanf(inp, "\n");
fclose(inp);
```

```
   printf("PROBLEM DATA: %d rows by %d columns", m, n);
   printf("  (Input file: %s)\n\n", inpname);
   if (PRINTINPUT) {
      printf("Objective coefficients:\n");
      for (i=1; i <= m; i++) {
         for (j=1; j<=n; j++) printf("%3d", c[i][j]);
         printf("\n");
      }
      printf("\nKnapsack constraints:\n");
      for (i=1; i <= m; i++) {
         for (j=1; j<=n; j++) printf("%3d", a[i][j]);
         printf(" <= %d\n", b[i]);
      }
      printf("\n");
   }

} /* end of readin */


void setparams()
{
   char iflag[3], bflag[3], oflag[3];

   FILE *cfg;
   int i, j;
   char dummy[20];

   if ((cfg = fopen("kblgap.cfg", "r")) == NULL) {
      printf( "Cannot open kblgap.cfg, defaults are in effect.\n");
   }
   else {
      fscanf(cfg, "%d %s\n", &INFINITY, dummy);
      fscanf(cfg, "%d %s\n", &ROOTMAMITLIM, dummy);
      fscanf(cfg, "%d %s\n", &MAMITLIM, dummy);
      fscanf(cfg, "%d %s\n", &ROOTSUBITLIM, dummy);
      fscanf(cfg, "%d %s\n", &SUBITLIM, dummy);
      fscanf(cfg, "%d %s\n", &SUBPATIENCE, dummy);
      fscanf(cfg, "%d %s\n", &MAXBRANCH, dummy);
      fscanf(cfg, "%d %s\n", &PRINTSTEPS, dummy);
      fscanf(cfg, "%d %s\n", &PRINTINPUT, dummy);
      fscanf(cfg, "%d %s\n", &PRINTOPTIM, dummy);
      fscanf(cfg, "%f %s\n", &ZEROTOL, dummy);
      fclose(cfg);
   }
```

```
if ((ROOTMAMITLIM <= 0 && ROOTSUBITLIM <= 0) || (MAMITLIM <= 0 && SUBITLIM <= 0)) {
   printf("ERROR: One of iter limits for upper bounding must be positive.\n");
   printf("  (Data: ROOTMAMITLIM = %d, ROOTSUBITLIM = %d\n",
          ROOTMAMITLIM, ROOTSUBITLIM);
   printf("              MAMITLIM = %d,      SUBITLIM = %d)\n", MAMITLIM, SUBITLIM);
   exit(1);
}

if (PRINTINPUT) strcpy(iflag,"YES"); else strcpy(iflag,"NO ");
if (PRINTSTEPS) strcpy(bflag,"YES"); else strcpy(bflag,"NO ");
if (PRINTOPTIM) strcpy(oflag,"YES"); else strcpy(oflag,"NO ");

printf("\n---PARAMETERS SET");
printf("----------------------------------------------------\n");
printf(
"                    MAM   SUB    PRINT FLAGS       GENERAL\n");
printf(
"  Root iter limit   %3d   %3d    Input     %s     Infinity    %d\n",
ROOTMAMITLIM, ROOTSUBITLIM, iflag, INFINITY);
printf(
"  Node iter limit   %3d   %3d    Branches  %s     Zero Tol    %6.4f\n",
MAMITLIM, SUBITLIM, bflag, ZEROTOL);
printf(
"  Patience          %3d          OptimSol  %s     Max Branch  %d\n",
SUBPATIENCE, oflag, MAXBRANCH);
printf("----------------------------------");
printf("----------------------------------\n");
printf(
"  Program Limits: Max Rows %d, Max Columns %d, Max Knapsack RHS %d.\n",
MAXM, MAXN, MAXB);
printf("----------------------------------");
printf("----------------------------------\n");

} /* end of setparams */
```

3.3 Upper Bounding Functions

Two procedures are run in serial to obtain an upper bound on the optimum objective of the GAP or any of its subproblems obtained by fixing some of the variables. First, a multiplier adjustment method (MAM) is called, followed by a subgradient algorithm. Both procedures solve a Lagrangian dual of the GAP for upper bounding. Associated Lagrangian relaxation is obtained by dualizing multiple-choice constraints.

`MAM()` and `subgradient()` share a common input list:

- For $j = 1, \dots, n$

$$\texttt{jfixed[j]} = \begin{cases} i & \text{if } x_{ij} = 1 \text{ and } x_{\ell j} = 0,\ \ell \neq i \\ 0 & \text{if } x_{1j}, \dots, x_{mj} \text{ are all free} \end{cases}$$

Let $S = \{j \mid \texttt{jfixed[j]} \neq 0\}$.

- Sum of objective values of fixed variables

$$\texttt{objfixed} = \sum_{j \in S} c_{kj}, \quad k = \texttt{jfixed[j]}$$

- For $i = 1, \ldots, m$

$$\texttt{bbar[i]} = b_i - \sum_{\{j \in S | k=i\}} a_{kj}, \quad k = \texttt{jfixed[j]}$$

- For $j = 1, \ldots, n$

  lambda[j] = initial multiplier associated with multiple-choice constraint j.

The outputs are the vector of final multipliers and the function value itself that gives the tightest upper bound obtained.

The following code for the MAM, described in detail in Karabakal, Bean, and Lohmann (1992), is based on the idea of steepest descent improvements per iteration in solving the Lagrangian dual. Computationally, it outperforms its predecessors suggested by Fisher, Jaikumar, and Van Wassenhove (1986) and Guignard and Rosenwein (1989).

```
int MAM(jfixed, objfixed, bbar, lambda)
int jfixed[], objfixed, bbar[], lambda[];
{
   int iter, i, j, k, rhs, istar, jstar, lastjstar, firsttime, row,
       p[MAXN+1], x[MAXM+1][MAXN+1], pos[MAXN+1],
       aknap[MAXN+1], xknap[MAXN+1], seq[MAXN+1],
       dual, cknap[MAXN+1], vknap, slack[MAXN+1],
       delta[MAXM+1][MAXN+1], delta1, delta2,
       imp, maximp, steplength, maxstep, bound;
   bool optcomp;
   void fbknap(), checkincumbent();

   /* initialize */
   lastjstar = 0;
   firsttime = YES;
   iter = 0;
   /* initialize dual */
   for (j=1, dual=0; j<=n; j++) if (jfixed[j]==0)
      dual += lambda[j];

   while (1) {

      /* initialize column totals */
      for (j=1; j<=n; j++) p[j] = 0;
```

```
/* solve m forward-backward knapsacks */
for (i=1; i<=m; i++) {
   k = 0;  /* k counts no of elements in knapsack */
   for (j=1; j<=n; j++)  if (jfixed[j] == 0)
      if (a[i][j] <= bbar[i]) {
         seq[++k] = j;
         cknap[k] = c[i][j] - lambda[j]; aknap[k] = a[i][j];
      }
   for (j=1; j<=n; j++) {
      x[i][j] = 0; delta[i][j] = INFINITY;
   }
   if (k > 0) {
      rhs = bbar[i];
      fbknap(k, rhs, cknap, aknap, xknap, &vknap, slack);
      if (firsttime) dual += vknap;
      for (j=1; j<=k; j++) {
         if (xknap[j]) { x[i][seq[j]] = 1; ++p[seq[j]]; }
         delta[i][seq[j]] = slack[j];
      }
   }
} /* end of m forward-backward knapsacks */

bound = objfixed + dual;
if (firsttime) {
   firsttime = NO;
   if (bound <= incumbent) return(bound);
}

optcomp = YES;          /* assume optimum completion */
for (j=1; j<=n; j++) if (jfixed[j]==0)
if ((p[j] != 1)) { optcomp = NO; break; }

if (optcomp) {          /* optimum completion, compute the position vector */
   for (j=1; j<=n; j++)  pos[j] = jfixed[j];
   for (j=1; j<=n; j++) if (jfixed[j] == 0)
      for (i=1; i<=m; i++)
         if (x[i][j] == 1) { pos[j] = i; break; }
   checkincumbent(pos);
   return(bound);
}
else {                  /* not optimum completion */
```

```
        /* compute the steepest descent */
        maximp = -INFINITY;
        for (j=1; j<=n; j++) if (jfixed[j] == 0)
           if ((p[j] != 1) && (j != lastjstar)) {
              delta1 = delta2 = INFINITY;
              for (i=1; i<=m; i++)
                 if ((p[j]==0) || ((p[j] > 1) && (x[i][j] == 1)))
                    if (delta[i][j] < delta1) {
                       delta2 = delta1; delta1 = delta[i][j];
                       row = i;
                    }
                    else {
                       if (delta[i][j] < delta2)  delta2 = delta[i][j];
                    }
              if ((delta2 != INFINITY) && (delta2 > 0)) {
                 imp = (p[j]==0) ? delta1 : (p[j]-2) * delta2 + delta1;
                 if (imp > maximp) {
                    maximp = imp; maxstep = delta2;
                    jstar = j;  istar = row;  steplength = delta2;
                 }
                 else
                    if ((imp == maximp) && (delta2 > maxstep)) {
                       maxstep = delta2;
                       jstar = j; istar = row; steplength = delta2;
                    }
              }
           } /* end of for j loop */
     } /* end of else not opt comp */

     if (maximp == -INFINITY) return(bound); /* zero step length */

     lastjstar = jstar;
     if (p[jstar] == 0) {
        dual -= delta[istar][jstar];
        lambda[jstar] -= steplength;
     }
     else {
        dual -= ( (p[jstar]-2) * steplength + delta[istar][jstar] );
        lambda[jstar] += steplength;
     }

     bound = objfixed + dual;
     if (bound <= incumbent) return(bound);
     else
    if (++iter > ((firstdangling == NULL) ? ROOTMAMITLIM : MAMITLIM))
       return(bound);

  } /* end of while (1) loop */
} /* end of MAM */
```

`MAM()` calls `fbknap()` function to solve knapsacks with a post-optimality analysis, i.e., with the calculation of the `slack` vector below.

Input:

- **nknap** = number of items in the knapsack problem,
- **rhs** = RHS capacity of the knapsack,
- **cknap[j]** = objective value of item `j`,
- **aknap[j]** = knapsack weight of item `j`,

Output

- Optimum solution

$$\textbf{xknap[j]} = \begin{cases} 1 & \text{if item j is selected in the optimum} \\ 0 & \text{otherwise} \end{cases}$$

- **vknap** = optimum objective value,
- Slacks ("Δ_j"s)

$$\texttt{slack[j]} = \begin{cases} \text{minimum } \delta \text{ such that } \textbf{cknap[j]}+\delta \text{ would cause} & \text{if } \textbf{xknap[j]} = 0 \\ \textbf{xknap[j]} = 1 \text{ in an alternative optimum} & \\ & \\ \text{minimum } \delta \text{ such that } \textbf{cknap[j]}-\delta \text{ would cause} & \text{if } \textbf{xknap[j]} = 1 \\ \textbf{xknap[j]} = 0 \text{ in an alternative optimum} & \end{cases}$$

See Karabakal, Bean, and Lohmann (1992) for the formulation of forward and backward dynamic programming recursions and for the derivation of the `slack` vector.

```
void fbknap(nknap, rhs, cknap, aknap, xknap, vknap, slack)
int nknap, rhs, aknap[], xknap[], cknap[], *vknap, slack[];
{
   int f[MAXN+1][MAXB], g[MAXN+1][MAXB];
   int soln[MAXN+1][MAXB], j, beta;

   /* forward recursions */
   for (beta=0; beta < aknap[1]; beta++) {
      f[1][beta] = 0; soln[1][beta] = 0;
   }
   for (beta=aknap[1]; beta <= rhs; beta++)
      if (cknap[1] > 0) {
         f[1][beta] = cknap[1];  soln[1][beta] = 1;
      }
      else {
         f[1][beta] = 0; soln[1][beta] = 0;
      }
```

```
for (j=2; j <= nknap; j++) {
   for (beta=0; beta < aknap[j]; beta++) {
      f[j][beta] = f[j-1][beta]; soln[j][beta] = 0;
   }
   for (beta=aknap[j]; beta <= rhs; beta++)
      if (f[j-1][beta] >= f[j-1][beta-aknap[j]] + cknap[j]) {
         f[j][beta] = f[j-1][beta];  soln[j][beta] = 0;
      }
      else {
         f[j][beta] = f[j-1][beta-aknap[j]] + cknap[j];
         soln[j][beta] = 1;
      }
}

/* determine optimum solution */
*vknap = f[nknap][rhs];
beta = rhs;  j = nknap;
while (j) {
   xknap[j] = soln[j][beta];
   if (soln[j][beta]) beta -= aknap[j];
   j--;
}

/* backward recursion */
for (beta=0; beta<=rhs; beta++)
   g[nknap][beta] = *vknap;

for (j=nknap-1; j >= 1; j--) {
   for (beta=rhs; beta >= rhs-aknap[j+1]+1; beta--)
      g[j][beta] = g[j+1][beta];
   for (beta=rhs-aknap[j+1]; beta >= 0; beta--)
      if (g[j+1][beta] < g[j+1][beta+aknap[j+1]] - cknap[j+1])
         g[j][beta] = g[j+1][beta];
      else
         g[j][beta] = g[j+1][beta+aknap[j+1]] - cknap[j+1];
}

/* compute slacks (Δ_j quantities) */
slack[1] = INFINITY;
if (xknap[1] == 0) {
   for (beta=aknap[1]; beta <= rhs; beta++)
      if (g[1][beta] - cknap[1] < slack[1])
         slack[1] = g[1][beta] - cknap[1];
}
else
   for (beta=0; beta<=rhs; beta++)
      if (g[1][beta] < slack[1]) slack[1] = g[1][beta];
   for (j=2; j<=nknap; j++) {
      slack[j] = INFINITY;
      if (xknap[j] == 0) {
         for (beta=aknap[j]; beta <= rhs; beta++)
            if (g[j][beta] - f[j-1][beta-aknap[j]] - cknap[j] < slack[j])
               slack[j] = g[j][beta] - f[j-1][beta-aknap[j]] - cknap[j];
      }
      else
         for (beta=0; beta <= rhs; beta++)
            if (g[j][beta] - f[j-1][beta] < slack[j])
               slack[j] = g[j][beta] - f[j-1][beta];
   }

} /* end of fbknap */
```

The subgradient() function implements the subgradient algorithm described in Fisher (1981) for solving Lagrangian duals of integer programming problems. The following step size is chosen for the GAP:

$$\texttt{stepsize} = \frac{u\,(L(\lambda) - \texttt{incumbent})}{\sum_{j=1}^{n}(1 - \sum_{i=1}^{m} x_{ij})^2}$$

```
int subgradient(jfixed, objfixed, bbar, lambda)
int jfixed[], objfixed, bbar[];
float lambda[];
{
   int iter = 0, badcount = 0, pos[MAXN+1],
       p[MAXN+1], minp[MAXN+1], x[MAXM+1][MAXN+1],
       aknap[MAXN+1], xknap[MAXN+1], seq[MAXN+1],
       i, j, k, rhs, bound;
   float u, dual, mindual = INFINITY,
         minlambda[MAXN+1], subgrad[MAXN+1],
         norm, stepsize, cknap[MAXN+1], vknap;
   bool optcomp;
   void knapdp(), lowerbound(), checkincumbent();

   if (firstdangling == NULL) u = 2.0; else u = 1.0;

   while (1) {

      /* initialize dual */
      for (j=1, dual=0; j<=n; j++) if (jfixed[j]==0)
         dual += lambda[j];

      /* initialize column totals */
      for (j=1; j<=n; j++) p[j] = 0;
```

```
/* solve m knapsacks */
for (i=1; i<=m; i++) {
   k = 0;  /* k counts no of elements in knapsack */
   for (j=1; j<=n; j++)  if (jfixed[j] == 0)
      if ((a[i][j] <= bbar[i]) && ( (c[i][j]-lambda[j]) > 0.0)) {
         seq[++k] = j;
         cknap[k] = c[i][j] - lambda[j]; aknap[k] = a[i][j];
      }
   for (j=1; j<=n; j++) x[i][j] = 0;
   if (k > 0) {
      rhs = bbar[i];
      knapdp(k, rhs, cknap, aknap, xknap, &vknap);
      dual += vknap;
      for (j=1; j<=k; j++)
         if (xknap[j]) { x[i][seq[j]] = 1; ++p[seq[j]]; }
   }
} /* end of knapsack solutions */

bound = objfixed + (int) dual;

optcomp = YES;                /* assume optimum completion */
for (j=1; j<=n; j++) if (jfixed[j]==0)
   if ((p[j] != 1)) { optcomp = NO; break; }

if (optcomp) {          /* optimum completion, compute the position vector */
   for (j=1; j<=n; j++)  pos[j] = jfixed[j];
   for (j=1; j<=n; j++) if (jfixed[j] == 0)
      for (i=1; i<=m; i++)
         if (x[i][j] == 1) { pos[j] = i; break; }
   checkincumbent(pos);
   return(bound);
}
else {                        /* not optimum completion */
   lowerbound(jfixed, x, bbar, p);
   if (bound <= incumbent)
      return(bound);
   else {                     /* neither optimum completion nor inferior */

      if (dual < mindual) {                /* solution improved */
         badcount = 0; mindual = dual;
         for (j=1; j<=n; j++) {
            minlambda[j] = lambda[j]; minp[j] = p[j];
         }
      }
      else {                               /* solution did not improve */

         if (++badcount == SUBPATIENCE) {

            /* restore last best solution found and continue from there */
            /* by halving the step size */
```

```
                badcount = 0; dual = mindual;
                for (j=1; j<=n; j++) {
                   lambda[j] = minlambda[j]; p[j] = minp[j];
                }
                u = u / 2;
             }
          }

          /* find a subgradient */
          for (j=1; j<=n; j++) if (jfixed[j] == 0)
             subgrad[j] = 1 - p[j];

          /* compute step size */
          norm = 0;
          for (j=1; j<=n; j++) if (jfixed[j] == 0)
             norm += subgrad[j] * subgrad[j];
          stepsize = u * (objfixed + dual - incumbent) / norm;

          if (stepsize > ZEROTOL) {  /* update multipliers */
             for (j=1; j<=n; j++) if (jfixed[j] == 0) {
                lambda[j] -= stepsize * subgrad[j];
             }


             if (++iter < ((firstdangling == NULL) ? ROOTSUBITLIM : SUBITLIM)) {
                continue;
             }
          }

          /* either zero step size or iteration limit exceeded */
          bound = objfixed + (int) mindual;
          for (j=1; j<=n; j++) lambda[j] = minlambda[j];
          return(bound);
       } /* else neither opt comp nor inferior */
    } /* else not opt comp */
 }  /* while (1) */

} /* end of subgradient */
```

The **knapdp()** function, which the subgradient algorithm calls for optimum knapsack solutions, is a linked-list implementation of the "Procedure P2" given by Toth (1980). It is a forward dynamic programming algorithm and uses the "further but cheaper"-type pruning technique to improve performance. Note that, under this pruning, the **slack** vector cannot be calculated and thus, this procedure is not suitable for solving the knapsack subproblems of the **MAM()**. Incidentally, the "Procedure P2" of Toth (1980) has a bug in it; it is corrected in the implementation here.

The input-output parameters defined for function **fbknap()** are also valid for **knapdp()**.

```
void knapdp(nknap, rhs, cknap, aknap, xknap, vknap)
int nknap, rhs, aknap[], xknap[];
float cknap[], *vknap;
{
   int L[MAXN+1][MAXB], val[MAXN+1][MAXB], prev[MAXN+1][MAXB],
       s[MAXN+1], h, k, j, y, xtemp, ptemp, mm;
   float F[MAXN+1][MAXB], ftemp;

   /* initialize for mm = 1 */
   L[1][0] = F[1][0] = val[1][0] = prev[1][0] = s[1] = 0;
   if ( (aknap[1] <= rhs) && (cknap[1] > 0) ) {
      L[1][1] = aknap[1]; F[1][1] = cknap[1];
      val[1][1] = 1; prev[1][1] = 0; s[1] = 1;
   }

   for (mm=2; mm <= nknap; mm++) {
      L[mm][0] = F[mm][0] = val[mm][0] = prev[mm][0] = 0;
      h = ( (s[mm-1] == 0) ? 0 : 1); k = j = 0; y = aknap[mm];
      while (1) {
         if ( L[mm-1][h] < y ) {
            if ( F[mm-1][h] > F[mm][k] ) {
               k++;
               L[mm][k] = L[mm-1][h]; F[mm][k] = F[mm-1][h];
               val[mm][k] = 0; prev[mm][k] = h;
            }
            if ( h == s[mm-1] ) break; else h++;
         }
         else {
            if ( L[mm-1][h] > y ) {
               ftemp = F[mm-1][j] + cknap[mm];
               if ( ftemp > F[mm][k] ) {
                  k++;
                  L[mm][k] = y; F[mm][k] = ftemp;
                  val[mm][k] = 1; prev[mm][k] = j;
               }
               y = L[mm-1][++j] + aknap[mm];
            }
            else {  /* L[mm-1][h] = y */
               ftemp = F[mm-1][j] + cknap[mm];
               xtemp = 1; ptemp = j;
               if (F[mm-1][h] > ftemp) {
                  ftemp = F[mm-1][h]; xtemp = 0; ptemp = h;
               }
               if (ftemp > F[mm][k]) {
                  k++;
                  L[mm][k] = y; F[mm][k] = ftemp;
                  val[mm][k] = xtemp; prev[mm][k] = ptemp;
               }
               y = L[mm-1][++j] + aknap[mm];
               if (h == s[mm-1]) break; else h++;
            }
         }
      } /* while (1) */
      /* step 3 */
      while (1) {
         if (y > rhs) break;
         else {
            ftemp = F[mm-1][j] + cknap[mm];
            if (ftemp > F[mm][k]) {
               k++;
               L[mm][k] = y;  F[mm][k] = ftemp;
               val[mm][k] = 1;  prev[mm][k] = j;
            }
            if (j == s[mm-1]) break;
            else
               y = L[mm-1][++j] + aknap[mm];
         }
      } /* while (1) */

      /* step 4 */
      s[mm] = k;
   } /* for (mm=2... */

   /* trace links back to recover solution */
   *vknap = F [nknap] [s[nknap]];
   k = s[nknap];
   for (mm=nknap; mm >= 1; mm--) {
      xknap[mm] = val[mm][k];
      k = prev[mm][k];
   }
} /* end of knapdp */
```

3.4 Lower Bounding Function (Heuristic)

Given a Lagrangian solution, x, that satisfies knapsack and integrality constraints, but violates multiple-choice constraints, the following heuristic attempts to obtain overall feasibility by modifying current assignments.

```
void lowerbound(jfixed, x, bbar, p)
int jfixed[], x[][MAXN+1], bbar[], p[];
{
   int pcopy[MAXN+1], xcopy[MAXM+1][MAXN+1], capacity[MAXM+1],
       pos[MAXN+1], aknap[MAXN+1], xknap[MAXN+1], seq[MAXN+1],
       i, j, k, rhs, istar, feasible;
   float minratio, temp, cknap[MAXN+1], vknap;
   void knapdp(), checkincumbent();

   /* copy p to pcopy, x to xcopy */
   for (j=1; j<=n; j++) {
      pcopy[j] = p[j];
      for (i=1; i<=m; i++)
         xcopy[i][j] = x[i][j];
   }
```

```
/* compute available knapsack capacities */

for (i=1; i<=m; i++) {
   capacity[i] = bbar[i];
   for (j=1; j<=n; j++) if (jfixed[j] == 0)
      if (xcopy[i][j] == 1) capacity[i] -= a[i][j];
}

/* Order the variables $x_{ij}$ such that $x_{ij} = 1$ and $p_j > 1$ in the current solution */
   in ascending $c_{ij}/a_{ij}$. Following this order, set $x_{ij} = 0$ and increase the slack
   space of knapsack $i$ by $a_{ij}$. Repeat until $p_j \leq 1$ for all $j$.

for (j=1; j<=n; j++) if (jfixed[j] == 0)
   while (pcopy[j] > 1) {
      minratio = INFINITY;
      for (i=1; i<=m; i++)
         if (xcopy[i][j] == 1) {
            temp = (float) c[i][j] / (float) a[i][j];
            if (temp < minratio) {
               minratio = temp; istar = i;
            }
         }
      xcopy[istar][j] = 0; --pcopy[j]; capacity[istar] += a[istar][j];
   }

/* Using only variables $x_{ij}$ with $p_j = 0$, solve all knapsack problems optimally */

for (i=1; i<=m; i++) {
   k = 0;                        /* k counts no of elements in knapsack */
   for (j=1; j<=n; j++)  if (jfixed[j] == 0)
      if ( (pcopy[j] == 0) && (a[i][j] <= capacity[i]) ) {
         seq[++k] = j;
         cknap[k] = c[i][j]; aknap[k] = a[i][j];
      }
   if (k > 0) {
      rhs = capacity[i];
      knapdp(k, rhs, cknap, aknap, xknap, &vknap);
      for (j=1; j<=k; j++)
         if (xknap[j] == 1) {
            xcopy[i][seq[j]] = 1; ++pcopy[seq[j]];
         }
   }
} /* end of knapsack solutions */
```

```
   feasible = 1;
   for (j=1; j<=n; j++) if (jfixed[j] == 0)
      if (pcopy[j] != 1) { feasible = 0; break; }
   if (feasible) {
      for (j=1; j<=n; j++)  pos[j] = jfixed[j];
      for (j=1; j<=n; j++) if (jfixed[j] == 0)
         for (i=1; i<=m; i++)
            if (xcopy[i][j] == 1) { pos[j] = i; break; }
      checkincumbent(pos);
   }

}  /* end of lowerbound */
```

3.5 Fathoming Functions

Fathoming functions are designed to initialize parameters and organize calling sequences. They return one if the node is fathomed, zero if the node is dangling. Fathomed nodes are cleared from the memory immediately, whereas dangling nodes are inserted into a priority queue ordered by upper bounds.

```
bool fathomroot()
{
   NODEPTR root;
   int jfixed[MAXN+1], i, j, max1, max2, cmin, cmax, bound;
   bool fathom, fathomnode();
   void putdangling();

   /* determine absolute lower and upper bounds */
   abslb = bound = 0;
   for (j=1; j<=n; j++) {
      cmin = INFINITY; cmax = -INFINITY;
      for (i=1; i<=m; i++) {
         if (c[i][j] < cmin) cmin = c[i][j];
         if (c[i][j] > cmax) cmax = c[i][j];
      }
      abslb += cmin;  bound += cmax;
   }
   incumbent = abslb;

   /* create the root node */
   if ((root = (NODEPTR) malloc(sizeof(NODE))) == NULL) {
      printf( "Cannot create root node."); exit(1);
   }
   root->nodeno = 0;
   root->parent = NULL;
   /* initialize multiplers by selecting the second maximum objective value */
   /* of the variables from each multiple-choice set */
   for (j=1; j<=n; j++) {
      max1 = max2 = -INFINITY;
      for (i=1; i<=m; i++) if (a[i][j] <= b[i]) {
         if (c[i][j] > max1) {
            max2 = max1;  max1 = c[i][j];
         }
         else
            if (c[i][j] > max2) max2 = c[i][j];
      }
      root->lambda[j] = (max2 == -INFINITY) ? 0 : max2;
   }

   if (PRINTSTEPS)
      printf("Node 0 : [%d, %d] ==> ", incumbent, bound);

   for (j=1; j<=n; j++) jfixed[j] = 0;   /* no variables are fixed at the root */
   fathom = fathomnode(root, jfixed);
   if (!fathom) putdangling(root);
   return(fathom);

} /* end of fathomroot */


bool fathomnode(bbnode, jfixed)
NODEPTR bbnode;
int jfixed[];
{
   int objfixed, bbar[MAXM+1], bound, i, j, mamiter, subiter,
        ilambda[MAXN+1], UB;
   float dlambda[MAXN+1];
   bool fathom;
   NODEPTR hold;
   int MAM(), subgradient();
   void printsol();

   if (bbnode->nodeno) {
      ++nbranch;
      mamiter = MAMITLIM; subiter = SUBITLIM;
   }
   else {
      mamiter = ROOTMAMITLIM; subiter = ROOTSUBITLIM;
   }

   /* add node's fixed var to jfixed */
   if (bbnode->nodeno > 0) jfixed[bbnode->jfix] = bbnode->ifix;
```

```
for (i=1; i<=m; i++) bbar[i] = b[i];
objfixed = 0;
for (j=1; j<=n; j++) if (jfixed[j]) {
   bbar[ jfixed[j] ] -= a[ jfixed[j] ] [j];
   objfixed += c[ jfixed[j] ] [j];
}

for (j=1; j<=n; j++) ilambda[j] = bbnode->lambda[j];
if (mamiter > 0)
   bound = MAM(jfixed, objfixed, bbar, ilambda);
if (bound > incumbent) {
   for (j=1; j<=n; j++) dlambda[j] = (float) ilambda[j];
   if (subiter > 0)
      bound = subgradient(jfixed, objfixed, bbar, dlambda);
   if (bound > incumbent) {
      for (j=1; j<=n; j++) bbnode->lambda[j] = (int) dlambda[j];
      bbnode->bound = bound;
   }
}
fathom = (bound <= incumbent) ? YES : NO;

if (PRINTSTEPS) {
   printf("[%d,%d]  ", incumbent, bound);
   if (fathom) printf("FATHOM\n"); else printf("DANGLING\n");
}

if(nbranch >= MAXBRANCH) {
   printf("Branch limit exceeded.\n");
   if (incumbent == abslb) {
      printf("No feasible solution found.");
   }
   else {
     printf("\nBest solution found = %d,  ", incumbent);
     if (bbnode->nodeno == 0) UB = bound;
     else {
        UB = (firstdangling == NULL) ? -INFINITY : firstdangling->bound;
        if (bbnode->ifix != m) {
           hold = bbnode->parent;
           if (hold->bound > UB)  UB = hold->bound;
        }
     }
     printf("Upper bound = %d", UB);
     printf("  (Error bound = %4.2f%%)\n",
        ( 100 * (float) (UB - incumbent) / (float) incumbent));
     printf("No of branches = %d\n", nbranch);
     if (PRINTOPTIM) printsol(xinc);
   }
   exit(0);
}
```

```
   /* correct jfixed */
   if (bbnode->nodeno > 0) jfixed[bbnode->jfix] = 0;
   return(fathom);

}  /* end of fathomnode */
```

3.6 Branch-and-Bound Function

A node selected from the list of dangling nodes must have a solution violating one or more multiple-choice sets. At the beginning, the only dangling node is the root. Select a violated multiple-choice set with largest multiplier. Since any feasible solution includes exactly one out of m variables from this constraint, create m branches (Bean 1984). At each branch, one variable is fixed at one, and the rest are automatically set to zero.

```
void BAB()
{
   NODEPTR cnode, kid, search, getdangling();
   int jfixed[MAXN+1], used[MAXM+1], i, j, jstar,
       maxlambda, serialno = 0;
   bool feasible, fathomnode();
   void putdangling(), checkincumbent();

   while (firstdangling != NULL) {
      cnode = getdangling();

      /* determine fixed and free vars */
      for (j=1; j<=n; j++) jfixed[j] = 0;
      search = cnode;
      while (search->parent != NULL) {
         jfixed[ search->jfix ] = search->ifix;
         search = search->parent;
      }

      /* select jstar = multiple choice set to branch (one having biggest multiplier) */
      maxlambda = -INFINITY; jstar = 0;
      for (j=1; j<=n; j++) if (jfixed[j] == 0)
         if (cnode->lambda[j] >= maxlambda) {
            maxlambda = cnode->lambda[j];
            jstar = j;
         }

      /* determine used knapsack capacities by fixed variables */
      for (i=1; i<=m; i++) used[i] = 0;
      for (j=1; j<=n; j++) if (jfixed[j] > 0)
         used[jfixed[j]] += a[jfixed[j]][j];
      if (jstar == 0) { /* hit the bottom, jfixed is nonzero */
         feasible = YES;
         for (i=1; i<=m; i++)
            if (used[i] > b[i]) { feasible = NO; break; }
         if (feasible) checkincumbent(jfixed);
      }
      else                                /* create m branches */
         for (i=1; i<=m; i++)
            if ((used[i]+a[i][jstar] <= b[i]) && cnode->bound > incumbent) {
               if ((kid = (NODEPTR) malloc(sizeof(NODE))) == NULL) {
                  printf( "Cannot create new branches.");
                  exit(1);
               }
               kid->nodeno = ++serialno;
               kid->parent = cnode;
               kid->ifix = i;  kid->jfix = jstar;
               for (j=1; j<=n; j++)
                  kid->lambda[j] = cnode->lambda[j];

               if (PRINTSTEPS)
                  printf("Node %d From %d: x(%d,%d) [%d,%d] ==> ",
                         kid->nodeno, cnode->nodeno, i, jstar,
                         incumbent, cnode->bound);

               if (fathomnode(kid,jfixed))
                  free((char *)kid);
               else                       /* node dangling */
                  putdangling(kid);       /* insert into queue */
            }
   } /* end of while */

} /* end of BAB */
```

3.7 Priority Queue Handling Functions

Three functions are needed for handling the priority queue of dangling nodes.

1. The function `putdangling()` puts a new node into the correct place so that the order of decreasing upper bounds is maintained.

```
void putdangling(newnode)
NODEPTR newnode;
{
   NODEPTR search, last;
   newnode->next = NULL;
   search = firstdangling;
   if (search == NULL) firstdangling = newnode;
   else {
      while (search != NULL)
         if (search->bound > newnode->bound) {
            last = search;
            search = search->next;
         }
         else break;
      newnode->next = search;
      if (search->nodeno == firstdangling->nodeno)
         firstdangling = newnode;
      else
         last->next = newnode;
   }
} /* end of putdangling */
```

2. The function `getdangling()` gets a dangling node with highest upper bound.

```
NODEPTR getdangling()
{
   NODEPTR hold = firstdangling;
   firstdangling = firstdangling->next;
   return(hold);
} /* end of getdangling */
```

3. The function `removeinferiors()` checks the list of dangling nodes for possible fathomes whenever a better incumbent is found.

```
void removeinferiors()
{
   NODEPTR last, prev;

   while (1)
      if (firstdangling == NULL) return;
      else {
         last = firstdangling; prev = NULL;
         while (last->next != NULL) {
            prev = last;  last = last->next;
         }
         if (last->bound <= incumbent) { /* delete last */
            free((char *)last);
            if (prev == NULL) { /* last was the only one */
               firstdangling = NULL; return;
            }
            else
               prev->next = NULL;
         }
         else return;
      }
} /* end of removeinferiors */
```

3.8 Main Function and Utilities

The function printsol() prints an input position vector (feasible solution) in an easy-to-visualize format.

```
void printsol(pos)
int pos[];
{
   int i, j, used[MAXM+1], cx=0;

   for (i=1; i<=m; i++) used[i] = 0;
   for (j=1; j<=n; j++) {
      i = pos[j];  used[i] += a[i][j]; cx += c[i][j];
   }
   printf("\n cx = %d\n    ",cx);
   for (j=1; j<=n; j++) printf("%1d", (j % 10));
   printf("  USED  RHS\n");
   for (i=1; i<=m; i++) {
      printf("%2d)", i);
      for (j=1; j<=n; j++)
         if (pos[j] == i) printf("1"); else printf(".");
      printf("  %4d  %3d\n", used[i], b[i]);
   }
} /* end of printsol */
```

The function **checkincumbent()** first computes the objective value of the input position vector (feasible solution). If it is better than incumbent, updates incumbent and fathoms inferior dangling nodes.

```
void checkincumbent(pos)

int pos[];
{
   int j, vpos = 0;
   void removeinferiors();

   for (j=1; j<=n; j++)  vpos += c[pos[j]] [j];
   if (vpos > incumbent) {
      incumbent = vpos;
      for (j=1; j<=n; j++) xinc[j] = pos[j];
      removeinferiors();
   }
} /* end of checkincumbent */


void main(argc, argv)
int argc;
char *argv[];
{
   char inpname[50];
   void readin(), BAB(), setparams(), printsol();
   bool fathomroot();

   setparams();

   if (argc == 2)
      strcpy(inpname, *++argv);
   else
      inpname[0] = '*';

   readin(inpname);

   /* Solve the root problem; if duality gap, continue with branch-and bound */
   if (!fathomroot())  BAB();

   printf("\nOptimum = %d\nNo of branches = %d\n",
               incumbent, nbranch);
   if (PRINTOPTIM) printsol(xinc);

} /* end of main */
```

References

[1] Bean, J. C., "A Lagrangian Algorithm for the Multiple Choice Integer Program," *Operations Research*, 32 (1984), 1185-1193.

[2] Fisher M. L., "The Lagrangian Relaxation Method for Solving Integer Programming Problems," *Management Science*, 27 (1981), 1–18.

[3] Fisher M. L., R. Jaikumar, and L. N. Van Wassenhove, "A Multiplier Adjustment Method for the Generalized Assignment Problem," *Management Science*, 32 (1986), 1095–1103.

[4] Guignard, M. and M. B. Rosenwein, "An Improved Dual Based Algorithm for the Generalized Assignment Problem," *Operations Research*, 37 (1989), 658–663.

[5] Karabakal, N., J. C. Bean, and J. R. Lohmann, "A Steepest Descent Multiplier Adjustment Method for the Generalized Assignment Problem," Technical Report 92-11, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, 1992.

[6] Martello, S. and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley and Sons, 1990.

[7] Toth, P., "Dynamic Programming Algorithms for the Zero-One Knapsack Problem," *Computing*, 25 (1980), 29–45.