INPE-5390-TAE/011

FINDING SOLUTIONS TO VALUE INDEPENDENT KNAPSACK PROBLEMS

Horacio Hideki Yanasse
Nei Yoshihiro Soma

INPE
São José dos Campos
Maio 1992

SECRETARIA DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-5390-TAE/011

FINDING SOLUTIONS TO VALUE INDEPENDENT KNAPSACK PROBLEMS

Horacio Hideki Yanasse
Nei Yoshihiro Soma

This paper was prepared during the first author's visit to the University of Sheffield, England

INPE
São José dos Campos
Maio 1992

CDU: 519.8

Key words: Knapsack; subset sum; linear diophantine equation; enumeration

FINDING SOLUTIONS TO VALUE INDEPENDENT KNAPSACK PROBLEMS[1]

***Horacio Hideki Yanasse**
****Nei Yoshihiro Soma[2]**

*INPE/LAC - São José dos Campos-SP - BRAZIL
**University of Sheffield - Sheffield - Inglaterra

Abstract

An enumeration scheme to find the k-best solutions to the value independent knapsack problem is presented. The scheme requires O(N(B-A(1))) of memory allocation where A(1), without loss of generality, is the smallest coefficient of the knapsack constraint. Memory requirements may be reduced to O(B-A(1)) with a slight deterioration in computational performance when retrieving the k-best solutions. Similar schemes are presented for the 0-1 case and when the variables have explicit upper bounds.

Keywords: knapsack, subset sum, linear diophantine equation, enumeration scheme

[1]Research partially funded by SERC/GRANT GR/F 68942

[2]On leave from CTA/ITA/SJCampos, SP, BRAZIL. Acknowledges financial support from CNPq grant 201237/88.1

INTRODUCTION

Consider the following knapsack problem (**KP**)

$$\textbf{Max} \quad C(1)X(1)+C(2)X(2)+...+C(N)X(N)$$

subject to

$$A(1)X(1)+A(2)X(2)+...+A(N)X(N) <= B \qquad (1)$$

$$X(i) >= 0 \text{ AND INTEGER FOR ALL i.}$$

KP is hard to solve; it is known to be a NP-hard problem[1].

Many research works can be found on knapsack or related problems[2-9]. For the particular case where C(i)/A(i) is constant for all i, called *the value independent knapsack problem* (**VIKP**), the performance of methods based on branch-and-bound deteriorates sharply since good bounds are hard to obtain. In this case, enumerative schemes (usually based on dynamic programming) are as good as any other solution method. Most dynamic programming type algorithms for the 1-best solution requires *O(NB)* of memory allocation although it is possible to have an improved implementation that requires only *O(B)* of memory requirements[10].

Algorithms for the k-best (k>1) solutions to **VIKP** can be derived using the current enumerative schemes. However, to the best of our knowledge, this has not been addressed explicitly in the literature. We present a few cases where such solutions might be of interest and therefore motivate our work.

The k-best solutions for the **VIKP** might be helpfull in the process of finding solutions to problems where the knapsack constraint (1) is one among possibly many other constraints of a problem.

Consider, for example, a cutting stock problem[11-13] where rectangular panels of different sizes have to be cut from a large board. Assume some cutting constraints (possibly due to the machine) must be satisfied: cuts have to be guillotine type and orthogonal, the cutting patterns have to be formed by strips from where cross cuts are made to obtain the final panels required.

A heuristic approach to generate feasible cutting patterns for this two dimensional problem could be a two stage procedure where good combination of strip widths are identified first and then, panels are fitted into these strips, the best way possible. The best combination of strip widths are obtained by solving a one-dimensional knapsack problem. Not only the best combination of widths but all the first k-best combinations are of interest to determine patterns. This is so because the best combination of strip widths may not necessarily produce the best pattern in terms of overall waste.

Another example arises again in the cutting stock context. Some of the approaches for solving the cutting stock problem require that the best cutting patterns or all cutting patterns have to be generated beforehand so that afterwards the best mix of patterns is determined. The best cutting patterns, in the one-dimensional case, are the best solutions of a **VIKP**.

A third situation where finding the k-best solutions of a **VIKP** might be of interest is when computer memory is limited and we are interested, for instance, in finding a solution to a linear diophantine equation. A branch-and-bound type algorithm might generate lots of branches before reaching a solution and most dynamic programming type algorithms requires $O(NB)$ of memory allocation. Both type of methods probably would be infeasible to run as they stand, in this limited memory computer, if N and/or B is large. However, if B were smaller, less memory is needed, at least for a dynamic programming type implementation.

Based on this last observation, consider the following example: "Find a solution to the linear diophantine equation:

$$637x_1+6475x_2+6847x_3+9752x_4+10000x_5+11785x_6+13042x_7 = 29269$$

$$x_i >= 0 \text{ and integer for all i."}$$

It is possible to define a surrogate inequality with smaller coefficients by simply scaling down the coefficients and the right hand side value by a common factor and conveniently rounding the coefficients to the nearest integer so that none of the solutions of the initial problem is lost.

For the previous equation one could obtain the following inequality by dividing the coefficients by 1000:

$$0x_1+6x_2+6x_3+9x_4+10x_5+11x_6+13x_7 <= 29$$

$$x_i >= 0 \text{ and integer for all i}$$

By enumerating the k-best solutions to this surrogate problem, one could verify their feasibility to the original problem and hence, obtain a solution:

solution number	solution	rhs	feasible
1	$x_3=1,x_5=1,x_7=1$	29	NO
2	$x_2=1,x_5=1,x_7=1$	29	NO
3	$x_4=1,x_5=2$	29	NO
4	$x_3=1,x_4=1,x_7=1$	28	NO
5	$x_3=1,x_6=2$	28	NO
6	$x_2=1,x_4=1,x_7=1$	28	YES!
.	...	..	..
k	...	..	..

For this example, the 6^{th}-best enumerated solution to the surrogate problem is a solution to the original equation.

Observe that a similar approach could be used to solve other related problems, for instance, equality constrained knapsack problems or an integer linear programming problem having a linear diphantine equation as one of its constraints.

The proposed enumeration scheme is presented next. It is not based on the traditional dynamic programming nor the branch and bound methods. Observe that a branch and bound method is obviously not adequate to find the k-best solutions (k>1). The method we propose is an extension of the Yanasse and Soma's algorithm[13] to solve the one-dimensional knapsack problem with equality constraint. It has advantages over previous dynamic programming implementations in terms of computer memory allocation and computational time.

THE ALGORITHM

Consider the following branching scheme:

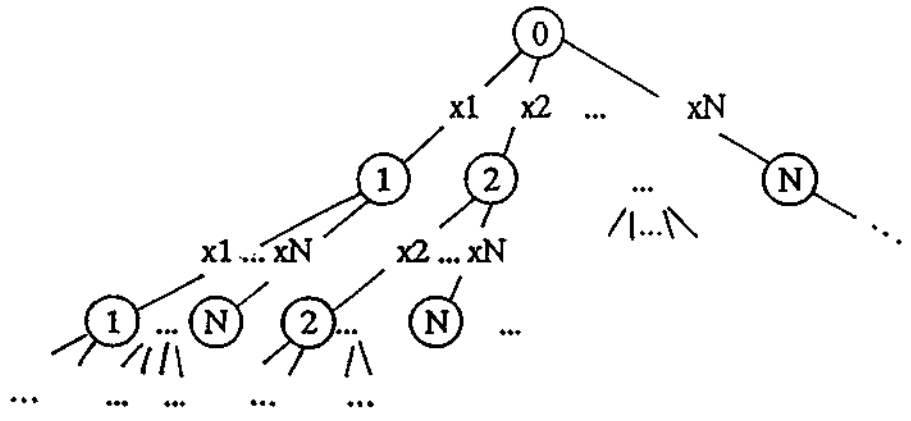

Figure 1
Enumeration tree

At each node an index, say J, is kept. Branching at each node is done according to the following:

> if J is the index kept at this node, new nodes are generated one for each one of the variables with index greater or equal to J, i.e. J, J+1,...,N. So, at most N nodes will be generated from each node.
>
> Stop branching at a node when the generation of another node implies the violation of constraint (1).

This scheme will enumerate all possible combinations of A(1)X(1)+ A(2)X(2)+...+A(N)X(N), for the different values that the X(i)'s can take. The desired solution is obtained by getting the k-best nodes achieved in the enumeration.

The algorithm is based on the previous scheme; care should be taken in enumerating the solutions in an efficient manner, that is, without repeating solutions. Each generated node has a value b associated with it, b < = B, corresponding to some linear combination of the coefficients. Different combinations might produce the same value b. To enumerate all solutions further branching from these nodes would be needed for all these combinations, starting from their respective indices. Duplication of computations is avoided by merging together all these nodes and using only the smallest index for branching. The additional branching required for the higher indices are implicitly enumerated due to the dominance of the smallest index. The recovering of the solutions is done by backtracking the node from where the branching was made to obtain the current node. This information is contained in the index (or indices) kept at each node.

Consider the following implementation where it is assumed that A(1) < = A(j) for all j.

Algorithm 1 for the k-best solutions

Define a B X N matrix M, and a B/A(1)-dimensional vector R. At the end of the enumeration, M(I,J) = 1 *if and only if* there exists a solution to A(1)X(1)+A(2)X(2)+...+A(J)X(J)=I, with X(J)>=1. Array R is used to keep indices while recovering the solutions.

Step 0 - *{Initialization}*
M(I,J):=0 for all I,J;
POINTER:=0;
M:=1;

Step 1 - WHILE POINTER <= B-1 DO
BEGIN
S:=M;
FOR J=S TO N DO
BEGIN
I:=A(J)+POINTER;
IF I<=B THEN M(I,J):=1;
END;
NAOTEM:=0;
WHILE (NAOTEM=0) AND (POINTER<B-1) DO
BEGIN
POINTER:=POINTER+1;
M:=0;
WHILE (M<=N) AND (NAOTEM=0) DO
BEGIN
M:=M+1;
IF M(POINTER,M)=1 THEN NAOTEM:=1;
END;
END;
END;

{generating the solutions}

Step 2 - POINTER:=B+1;
NUM:=0;
COUNT:=0;

Step 3 - POINTER:=POINTER-1;
IF POINTER=0 THEN
BEGIN
WRITE 'There is only 'NUM' solutions to this
problem';
STOP;
END;
REF:=POINTER;
S:=N;
FOR I=1 to N DO X(I):=0;
WHILE (S>0) AND (M(POINTER,S)=0) DO S:=S-1;
IF S=0 THEN RETURN TO **Step 3**;

```
        X(S):=X(S)+1;
        COUNT:=COUNT+1;
        R(COUNT):=S;

Step 4 - REF:=REF-A(R(COUNT));
        WHILE REF>0 DO
        BEGIN
            I:=R(COUNT);
            WHILE M(REF,I)=0 DO I:=I-1;
            X(I):=X(I)+1;
            COUNT:=COUNT+1;
            R(COUNT):=I;
            REF:=REF-A(R(COUNT));
        END;
        NUM:=NUM+1;
        WRITE solution;
        {solution number 'num' is given in the X array}

Step 5 - WHILE (NUM<k) DO
        BEGIN
            X(R(COUNT)):=X(R(COUNT))-1;
            REF:=REF+A(R(COUNT));
            I:=R(COUNT)-1;
            WHILE (I>0) AND M(REF,I)=0 DO I:=I-1;
            IF I>0 THEN
            BEGIN
                R(COUNT):=I;
                X(I):=X(I)+1;
                RETURN TO Step 4;
            END
            ELSE
            BEGIN
                COUNT:=COUNT-1;
                IF COUNT>0 THEN RETURN TO Step 5
                    ELSE RETURN TO Step 3;
            END;
        END;
```

Observe that this algorithm can be implemented using *O(N(B-A(1)))* of memory allocation requirements for the first A(1)-1 rows of matrix M carry no information.

Variations of this algorithm can be implemented using a similar scheme and requiring less memory allocation. This is not obtained freely but in exchange of possibly more computations to retrieve the solutions as compared with algorithm 1.

Care on memory allocation requirements is of particular interest for algorithms implemented on small computers where limitations on memory requirements might be quite restrictive.

Algorithm 2 is an example of a possible implementation of algorithm 1 where only the smallest index from where a node was generated is kept. Therefore, only an array of size B-A(1) is required, instead of a matrix of size (B-A(1)) X N. The solutions recovering operation has to be exhaustive since some valuable information are not stored.

Algorithm 2 for the k-best solutions

Define a B-vector M and a B/A(1)-dimensional vector R. At the end of the enumeration M(I)=0 implies that there is no solution for A(1)X(1)+A(2)X(2) +... +A(N)X(N)=I; M(I)=J, with J not equal to zero, implies that there is at least one solution to the previous equation with X(J)>=1 and possibly others with X(S)>=1 where N>=S>J. So, J is the smallest index for which a solution exists. The vector R serves the same purpose as in algorithm 1.

Step 0 - *{Initialization}*

```
M(I):=0 for all I;
POINTER:=0;
M:=1;
```

Step 1 - *{Forward labeling}*

```
WHILE POINTER<=B-1 DO
BEGIN
    S:=M;
    FOR J=S TO N DO
    BEGIN
        I:=A(J)+POINTER;
        IF I<=B THEN
            IF (M(I)=0) OR (M(I)>J) THEN M(I):=J
    END;
    NAOTEM:=TRUE;
    WHILE (NAOTEM) AND (POINTER<B-1) DO
    BEGIN
        POINTER:=POINTER+1;
        IF M(POINTER)>0 THEN
        BEGIN
            M:=M(POINTER);
```

```
                        NAOTEM:=FALSE;
                    END;
                END;
            END;

            {Generating the solutions}

Step 2 -    POINTER:=B;
            NUM:=0;
            COUNT:=0;
            NAOACHOU:=TRUE;

Step 3 -    WHILE (NAOACHOU) AND (POINTER>0) DO
            BEGIN
                IF M(POINTER)=0 THEN POINTER:=POINTER-1;
                    ELSE NAOACHOU:=FALSE;
            END;
            IF POINTER=0 THEN
            BEGIN
                WRITE 'There is only 'NUM' solutions to
                            this problem';
                STOP;
            END;
            REF:=POINTER;
            S:=N;
            FOR I=1 TO N DO X(I):=0;
            WHILE (S>=M(POINTER)) DO
            BEGIN
                J:= POINTER-A(S);
                IF (J>0) AND ((M(J)>0) AND ((M(J)<=S) THEN
                BEGIN
                    X(S):=X(S)+1;
                    COUNT:=COUNT+1;
                    R(COUNT):=S;
                    POINTER:=J;
                END
                ELSE
                BEGIN
                    IF J=0 THEN
                    BEGIN
                        X(S):=X(S)+1;
                        NUM:=NUM+1;
                        COUNT:=COUNT+1;
                        R(COUNT):=S;
                        WRITE solution;
                        {solution 'num' is given in the X array}
                        IF NUM=k THEN STOP;
                        CONTINUA:=0;
                        WHILE (COUNT>=1) AND (CONTINUA=0) DO
                        BEGIN
                            X(R(COUNT)):=X(R(COUNT))-1;
                            J:=J+A(R(COUNT));
                            S:=R(COUNT);
                            COUNT:=COUNT-1;
                            IF (S>M(J)) CONTINUA:=1;
                        END;
                    END
                    ELSE
```

```
                BEGIN
                    J:=J+A(S);
                    IF (S<=M(J)) THEN
                    BEGIN
                        CONTINUA:=0;
                        WHILE (COUNT>=1) AND
                        (CONTINUA=0) DO
                        BEGIN
                            X(R(COUNT)):=X(R(COUNT))-1;
                            J:=J+A(R(COUNT));
                            S:=R(COUNT);
                            COUNT:=COUNT-1;
                            IF (S>M(J)) CONTINUA:=1;
                        END;
                    END;
                END;
                S:=S-1;
                POINTER:=J;
            END;
        END;
        POINTER:=REF-1;
        COUNT:=0;
        NAOACHOU:=TRUE;
        RETURN TO Step3;
```

This algorithm needs *O(B-A(1))* of memory allocation although in our implementation we used a vector of size B. Observe again that the first A(1)-1 elements of vector M are always zero. It is possible to implement variations of the algorithm with additional memory requirements (but still of the same order) in an attempt to speed up the recovering of the solutions. We could keep, for instance, another array with the largest indices for which a solution exists or the total number of indices for which a solution exists for each value of the right hand side.

If the interest is solely on finding a single solution (1-best) to the **VIKP**, algorithm 2 is a better implementation than algorithm 1 or any other dynamic programming based algorithm that these authors area aware of, in terms of memory requirements and number of computer operations. Observe that the computational complexity of this algorithm is limited by *O(N(B-A(1))* in this case.

The proposed enumeration scheme can be easily modified to deal with the 0-1 case.

The 0-1 case

A simple adjustment can be made to solve problems where the variables are limited to 0 or 1. If J is the index kept at a node we now start branching from index J+1 instead of J. In algorithm 1 **Steps 0, 1** and **4** are replaced by:

Step 0' -
```
{initialization}
M(I,J):=0 for all I,J;
POINTER:=0;
M:=0;
```

Step 1' -
```
WHILE POINTER <= B-1 DO
BEGIN
        S:=M;
        FOR J=S+1 TO N DO
        BEGIN
                I:=A(J)+POINTER;
                IF I<=B THEN M(I,J):=1;
        END;
        NAOTEM:=0;
        WHILE (NAOTEM=0) AND (POINTER<B-1) DO
        BEGIN
                POINTER:=POINTER+1;
                M:=0;
                WHILE (M<=N) AND (NAOTEM=0) DO
                BEGIN
                        M:=M+1;
                        IF M(POINTER,M)=1 THEN NAOTEM:=1;
                END;
        END;
END;
```

Step 4' -
```
REF:=REF-A(R(COUNT));
WHILE REF>0 DO
BEGIN
        I:=R(COUNT)-1;
        WHILE M(REF,I)=0 DO I:=I-1;
        X(I):=1;
        COUNT:=COUNT+1;
        R(COUNT):=I;
        REF:=REF-A(R(COUNT));
END;
NUM:=NUM+1;
WRITE solution;
{solution number 'num' is given in the X array}
```

We refer to the above modified version as algorithm 1'.

Algorithm 2 can similarly be adjusted to solve 0-1 problems and obtain algorithm 2'.

Algorithm 2' *(0-1 case)*

Step 0 - *{Initialization}*

```
M(I):=0 for all I;
POINTER:=0;
M:=0;
```

Step 1 - *{Forward labeling}*

```
WHILE POINTER<=B-1 DO
BEGIN
        S:=M+1;
        FOR J=S TO N DO
        BEGIN
                I:=A(J)+POINTER;
                IF I<=B THEN
                        IF (M(I)=0) OR (M(I)>J) THEN M(I):=J
        END;
        NAOTEM:=TRUE;
        WHILE (NAOTEM) AND (POINTER<B-1) DO
        BEGIN
                POINTER:=POINTER+1;
                IF M(POINTER)>0 THEN
                BEGIN
                        M:=M(POINTER);
                        NAOTEM:=FALSE;
                END;
        END;
END;
```

{Generating the solutions}

Step 2 -

```
POINTER:=B;
NUM:=0;
COUNT:=0;
NAOACHOU:=TRUE;
```

Step 3 -

```
WHILE (NAOACHOU) AND (POINTER>0) DO
BEGIN
        IF M(POINTER)=0 THEN POINTER:=POINTER-1;
                ELSE NAOACHOU:=FALSE;
END;
IF POINTER=0 THEN
BEGIN
        WRITE 'There is only 'NUM' solutions to
                        this problem';
        STOP;
END;
REF:=POINTER;
S:=N
FOR I=1 TO N DO X(I):=0;
WHILE (S>=M(POINTER)) DO
BEGIN
```

```
        J:= POINTER-A(S);
        IF ((J>0) AND (M(J)>0) AND (M(J)<S)) THEN
        BEGIN
                X(S):=1;
                COUNT:=COUNT+1;
                R(COUNT):=S;
        END
        ELSE
        BEGIN
                IF J=0 THEN
                BEGIN
                        X(S):=1;
                        NUM:=NUM+1;
                        COUNT:=COUNT+1;
                        R(COUNT):=S;
                        WRITE solution;
                        {solution 'num' is given in the X array}
                        IF NUM=k THEN STOP;
                        CONTINUA:=0;
                        WHILE (COUNT>1) AND (CONTINUA=0) DO
                        BEGIN
                                X(R(COUNT)):=0;
                                J:=J+A(R(COUNT));
                                S:=R(COUNT);
                                COUNT:=COUNT-1;
                                IF (S>M(J)) CONTINUA:=1;
                        END;
                END
                ELSE
                BEGIN
                        J:=J+A(S);
                        IF (S<=M(J)) THEN
                        BEGIN
                                CONTINUA:=0;
                                WHILE (COUNT>=1) AND
                                (CONTINUA=0) DO
                                BEGIN
                                        X(R(COUNT)):=0;
                                        J:=J+A(R(COUNT));
                                        S:=R(COUNT);
                                        COUNT:=COUNT-1;
                                        IF (S>M(J)) CONTINUA:=1;
                                END;
                        END;
                END;
        END;
        S:=S-1;
        POINTER:=J;
END;
POINTER:=REF-1;
COUNT:=0;
NAOACHOU:=TRUE;
RETURN TO Step3;
```

The general problem with explicit bounds on the variables could be solved using algorithm 1 and afterwards, deleting the solutions which do not satisfy the upper limit constraints. We suggest instead an improved procedure.

It is possible to implement a modified version of algorithm 1 which avoids enumerating some (not all) solutions that violate the bounds on the variables and therefore, potentially decreasing the number of infeasibility checks in the retrievement of the solutions. Let the element M(I,J) of matrix M be used to keep the following information:

if M(I,J) = 0 then there is no solution to A(1)X(1)+ A(2)X(2)+ +A(J)X(J)=I, with X(J)>=1.

if M(I,J) = q > 0 , q integer, then there is a solution to A(1)X(1)+A(2)X(2)+ +A(J)X(J)=I, with X(J)>=q.

So, when branching at a node a check is made whether the upper limit condition of the leading variable (the one corresponding to the smallest index) is being satisfied. This modified algorithm is presented in the appendix.

Similar modifications could be made to algorithm 2 to handle problems with bounded variables. Recall that M(I)=J>0 in algorithm 2 implies that there is a solution for A(1)X(1)+ A(2)X(2) +... +A(N)X(N)=I; with X(J)>=1. To avoid enumerating solutions which violates the upper bound constraints on the variables we need to know how many A(J)'s were used so far up to that point. It is possible to carry this information in another array, say Y, the same size as array M. The memory allocation is duplicated but the order of the memory requirements for this modified algorithm is the same.

There is no need to duplicate the memory requirements if a different enumeration order (as compared with the previous algorithms) is used. Algorithm 3 to be presented next requires the same amount of memory as algorithm 2.

Let us perform the enumeration in the following way. At each node in the enumeration tree an index, say J, is kept. The branching scheme will generate new nodes from each node, according to the variables J+1,J+2,...,N. For each one of these variables U(*) nodes are generated (U(*) is the upper bound limit on variable *) if they do not violate constraint (1). With this we avoid the need of having to keep how many A(J)'s were used so far to check whether the upper bound limit is being satisfied. The implementation is again done in an efficient manner avoiding repeated branching.

Algorithm 3 *(Bounded variables)*

Step 0 - *{Initialization}*

```
M(I):=0 for all I;
POINTER:=0;
M:=0;
```

Step 1 - *{Forward labeling}*

```
WHILE (POINTER<=B-1) DO
BEGIN
    S:=M+1;
    FOR J=S TO N DO
    BEGIN
        K:=1;
        I:=POINTER;
        GOOD:=TRUE;
        WHILE (K<=U(J) AND GOOD) DO
        BEGIN
            I:=I+A(J);
            IF (I<=B) THEN
                IF (M(I)=0) OR (M(I)>J) THEN
                    M(I):=J;
                ELSE GOOD:=FALSE;
            ELSE GOOD:=FALSE;
            K:=K+1;
        END;
    END;
    NAOTEM:=TRUE;
    WHILE (NAOTEM) AND (POINTER<=B-1) DO
    BEGIN
        POINTER:=POINTER+1;
        IF (M(POINTER)>0) THEN
        BEGIN
            M:=M(POINTER);
            NAOTEM:=FALSE;
        END;
    END;
END;
```

```
        {Generating the solutions}

Step 2 -  POINTER:=B+1;
          NUM:=0;
          COUNT:=0

Step 3 -  POINTER:=POINTER-1;
          IF (POINTER=0) THEN
          BEGIN
                WRITE 'There is only 'NUM' solutions to
                            this problem';
                STOP;
          END;
          IF (M(POINTER)=0) THEN RETURN TO Step 3
          S:=N;
          FOR I=1 TO N DO X(I):=0;
          WHILE (S>=M(POINTER)) DO
          BEGIN
                J:=POINTER-A(S);
                IF (J>0) AND ((M(J)>0) AND (M(J)<=S)) THEN
                BEGIN
                      IF (X(S)<U(S)) THEN
                      BEGIN
                            X(S):=X(S)+1;
                            COUNT:=COUNT+1;
                            R(COUNT):=S;
                            POINTER:=J;
                      END
                      ELSE
                      BEGIN
                            J:=J+A(S);
                            IF (S<=M(J)) THEN
                            BEGIN
                                  CONTINUA:=0;
                                  WHILE (COUNT>=1) AND
                                  (CONTINUA=0) DO
                                  BEGIN
                                        X(R(COUNT)):=X(R(COUNT))-1;
                                        J:=J+A(R(COUNT));
                                        S:=R(COUNT);
                                        COUNT:=COUNT-1;
                                        IF (S>M(J)) CONTINUA:=1;
                                  END;
                            END;
                            S:=S-1;
                            POINTER:=J;
                      END;
                END
                ELSE
                BEGIN
                      IF (J=0) THEN
                      BEGIN
                            IF (X(S)<U(S)) THEN
                            BEGIN
                                  X(S):=X(S)+1;
                                  NUM:=NUM+1;
                                  COUNT:=COUNT+1;
```

```
                        R(COUNT):=S;
                        WRITE solution;
                    {solution 'num' is given in the X array}
                        IF NUM=k THEN STOP;
                        CONTINUA:=0;
                        WHILE (COUNT>=1) AND
                        (CONTINUA=0) DO
                        BEGIN
                            X(R(COUNT)):=X(R(COUNT))-1;
                            J:=J+A(R(COUNT));
                            S:=R(COUNT);
                            COUNT:=COUNT-1;
                            IF (S>M(J)) CONTINUA:=1;
                        END;
                    END
                    ELSE
                    BEGIN
                        J:=J+A(S);
                        IF (S<=M(J)) THEN
                        BEGIN
                            CONTINUA:=0;
                            WHILE (COUNT>=1) AND
                            (CONTINUA=0) DO
                            BEGIN
                                X(R(COUNT)):=
                                    X(R(COUNT))-1;
                                J:=J+A(R(COUNT));
                                S:=R(COUNT);
                                COUNT:=COUNT-1;
                                IF (S>M(J)) CONTINUA:=1;
                            END;
                        END
                    END;
                END
                ELSE
                BEGIN
                    J:=J+A(S);
                END;
                S:=S-1;
                POINTER:=J;
            END;
        END;
        COUNT:=0;
        RETURN TO Step3;
```

This modified enumeration order could be used to produce similar algorithms equivalent to algorithms 1 and 2 (and also 4, in the appendix) with comparable performances. The 0-1 case, is the limit case where both enumeration orders are indistinguishable.

An efficient generation of the 1-best solution for the bounded variables case can be obtained replacing Steps 2 and 3 of algorithm 3 with the following:

{generating 1 solution}

Step 2' -

```
POINTER:=B;
FOR I=1 TO N DO X(I):=0;
WHILE (M(POINTER)=0) DO POINTER:=POINTER-1;
INDEX:=M(POINTER);
K:=1;
NEXT:=0;
I:=POINTER;
WHILE ((K<U(INDEX)) AND (NEXT=0)) DO
BEGIN
        I:=I-A(INDEX);
        IF (I=0) THEN
        BEGIN
                X(INDEX):=K;
                NEXT:=1;
        END
        ELSE
        BEGIN
                IF (M(I)<INDEX) THEN
                BEGIN
                        X(INDEX):=K;
                        K:=1;
                        INDEX:=M(I);
                END
                ELSE
                        K:=K+1;
        END;
END;
```

We refer to this modified version as algorithm 3'.

Some limited computational tests were performed and are presented next.

COMPUTATIONAL EXPERIMENTS

Some computational tests were made comparing algorithms 1' and 2'.

For all test problems, all the coefficients A(i)'s of the knapsack constraint were randomly generated in the interval [1,500]. We were interested in observing the effect on computational time when the value of the right hand side changes, the number of variables in the problem increases and the number k of the best solutions desired increases. For each set of parameters tested, a fixed sample size of 50 was used.

Algorithms 1' and 2' were implemented in Turbo-Pascal version 5.5 on a Toshiba Lap-Top 1200HB (performance index equal twice of an IBM PC/XT). Since this Turbo-Pascal version can handle only 64 Kbytes of addressable space, the maximum right hand side value considered in the tests was 1000 which was set taking into consideration the memory requirements of algorithm 1'. All processing times were measured in seconds.

For the first set of tests, we varied the number k of the k-best solutions required. k was set to 10,20,...,90,100. The right hand side value was fixed in 1000. The number of variables was fixed in 50. In Figure 2 we present the results for algorithms 1' and 2'. Algorithm 2' outperformed algorithm 1' in all cases.

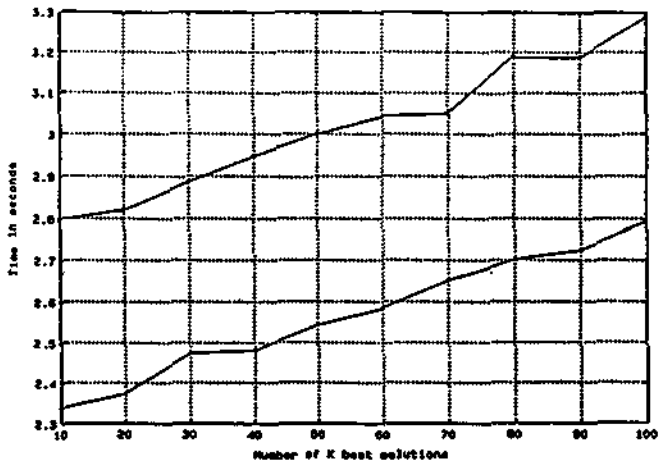

Figure 2
Processing times with varying k's for algorithms 1' and 2'

In the second set of problems we varied the number of variables n. n took the values 10,11,12,...,50. The right hand side and k were fixed in 100. The results are presented in Figure 3. Again, algorithm 2' outperformed algorithm 1'.

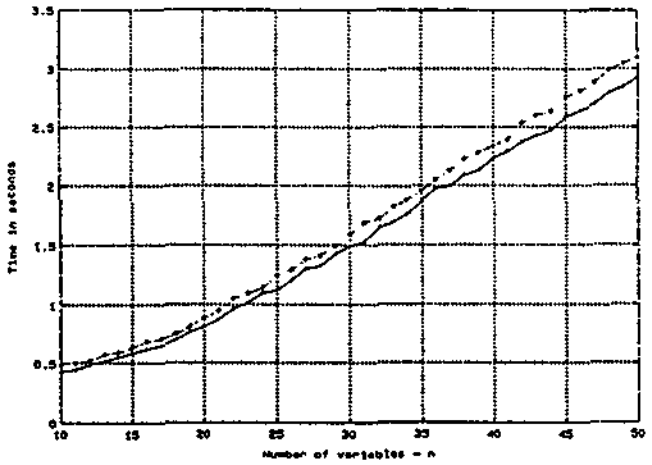

Figure 3
Processing times for algorithms 1' and 2' when n varies

In the last set of tests we varied the right hand side values. They took the values 500,600,...,1000. The number of variables and k were fixed in 50 and 100, respectively. The results are presented in Figure 4. Once again, algorithm 2' outperformed algorithm 1'.

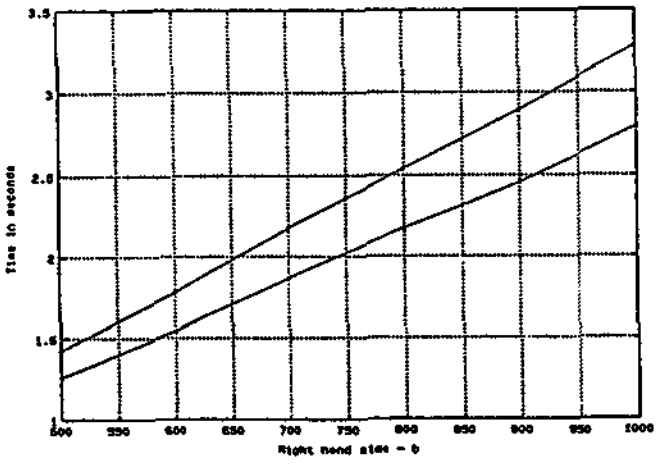

Figure 4
Processing times of algorithm 1' and 2' with varying B's

From the test results, we can see that the times obtained from algorithm 2' is slightly inferior to algorithm 1'. The forward enumeration is responsible for most of the processing time as N increases as can be seen comparing Figure 5 and Figure 3. The forward enumeration of algorithm 1' was always more time consuming than algorithm 2'.

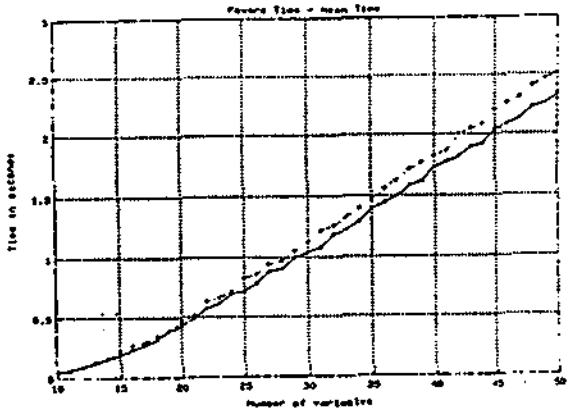

Figure 5
Forward enumeration contribution to the processing times

Retrieving the solutions demands an amount of time which depends on k and N, but is quite insensitive with the value of the right hand side. Its contribution to the overall computer processing time sharply decreases (percentagewise) as the size of the problem increases.

The backward retrievement of the solutions in algorithm 1' should be faster than in algorithm 2' for N and/or k large. For the range of values N tested, the backward retrievement of algorithm 2' was faster than of algorithm 1' for the smaller values of N, becoming increasingly comparable when N was increased (see Figure 6).

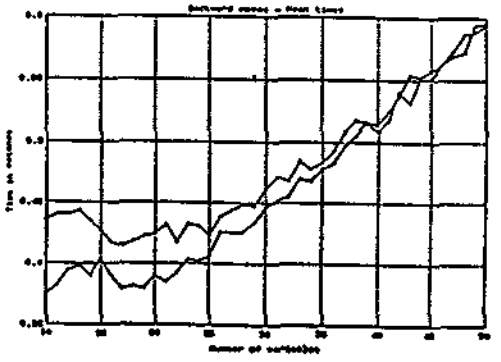

Figure 6
Backward retrievement of solutions for algorithm 1' and 2'

It is interesting to observe that from the limited test results algorithm 2' was always faster than algorithm 1'. Obviously we cannot guarantee that this will be the general case specially for larger values of k. From Figure 4, we might speculate that there are indications showing that for larger values of N, algorithm 2' might stay faster than algorithm 1' since the gap between the two lines seems to increase with N. That might be so, because although the retrievement of solutions in algorithm 1' is faster than in algorithm 2' for N large, it is not fast enough to compensate its slower forward enumeration.

We expect a comparable behaviour for the other cases we considered for all algorithms are similar in nature.

SUMMARY AND CONCLUSIONS

Algorithms for finding the k-best solutions to the value independent knapsack problem were presented, including the 0-1 and the general bounded variables cases.

Special care was taken in proposing alternative algorithms which require smaller amounts of computer memory. Limited computational tests showed that for the 0-1 case the "alternative" algorithm has an improved computer runtime as compared with its corresponding "original" algorithm if the number k of the k-best solutions required is kept small.

Implementations were suggested for the 1-best case which are marginally better than previously known dynamic programming type procedures with regard to memory allocation and/or computational time.

The computational tests showed that the major contributor to the overall computational time is the forward enumeration when N and/or B is large. This gives encouragement to

the idea of exploring further the approach presented in the introduction about scaling down the original data to solve certain problems. This is currently being investigated by these authors.

Appendix - An algorithm for bounded variables

It is assumed that X(I) < = U(I) for I = 1,...,N and the U(I)'s satisfy 1 < = U(I) < = [B/A(I)] for all I, where [s] denotes the largest integer less or equal to s.

Algorithm 4 *(bounded variables)*

Step 0 -

```
{Initialization}
M(I,J):=0 for all I,J;
POINTER:=0;
FOR J=1 TO N DO
BEGIN
        I:=A(J)+POINTER;
        IF I<=B THEN M(I,J):=1;
END;
NAOTEM:=0;
WHILE (NAOTEM=0) AND (POINTER<B-1) DO
BEGIN
        POINTER:=POINTER+1;
        M:=0;
        WHILE (M<=N) AND (NAOTEM=0) DO
        BEGIN
                M:=M+1;
                IF M(POINTER,M)=1 THEN NAOTEM:=1;
        END;
END;
```

Step 1 -

```
WHILE POINTER <= B-1 DO
BEGIN
        IF (M(POINTER,M)<U(M)) THEN S:=M;
                ELSE S:=M+1;
        FOR J=S TO N DO
        BEGIN
                I:=A(J)+POINTER;
                IF I<=B THEN
                        IF (J>M) THEN M(I,J):=1;
                                ELSE M(I,J):=M(POINTER,M)+1;
        END;
        NAOTEM:=0;
```

```
        WHILE (NAOTEM=0) AND (POINTER<B-1) DO
        BEGIN
            POINTER:=POINTER+1;
            M:=0;
            WHILE (M<=N) AND (NAOTEM=0) DO
            BEGIN
                M:=M+1;
                IF (M(POINTER,M)>=1) THEN NAOTEM:=1;
            END;
        END;
END;

{generating the solutions}
```

Step 2 -

```
POINTER:=B+1;
NUM:=0;
COUNT:=0;
```

Step 3 -

```
POINTER:=POINTER-1;
IF POINTER=0 THEN
BEGIN
    WRITE 'There is only 'NUM' solutions to this
            problem';
    STOP;
END;
REF:=POINTER;
S:=N;
FOR I=1 to N DO X(I):=0;
WHILE (S>0) AND (M(POINTER,S)=0) DO S:=S-1;
IF (S=0) THEN RETURN TO Step 3;
X(S):=X(S)+1;
COUNT:=COUNT+1;
R(COUNT):=S;
```

Step 4 -

```
REF:=REF-A(S);
NAOTEM:=0;
WHILE ((REF>0) AND (NAOTEM=0)) DO
BEGIN
    I:=S;
    WHILE (I>0 AND (M(REF,I)=0 OR X(I)=U(I))) DO
        I:=I-1;
    IF (I>0) THEN
    BEGIN
        X(I):=X(I)+1;
        COUNT:=COUNT+1;
        R(COUNT):=I;
        REF:=REF-A(R(COUNT));
        S:=R(COUNT);
    END
    ELSE
    BEGIN
        REF:=REF+A(S);
        S:=S-1;
        NAOACHOU:=TRUE;
        WHILE (NAOACHOU) DO
        BEGIN
            WHILE ((S>0) AND (M(REF,S)=0))
                DO S:=S-1;
```

```
                    IF (S=0) THEN
                    BEGIN
                        IF (COUNT>1) THEN
                        BEGIN
                            COUNT:=COUNT-1;
                            X(R(COUNT)):=X(R(COUNT))-1;
                            REF:=REF+A(R(COUNT));
                            S:=R(COUNT)-1;
                        END
                        ELSE
                        BEGIN
                            NAOTEM:=1;
                            NAOACHOU:=FALSE;
                        END;
                    END
                    ELSE
                        NAOACHOU:=FALSE;
                END;
            END;
        END;
        IF (NAOTEM=1) GOTO Step 3;
        NUM:=NUM+1;
        WRITE solution;
        {solution number 'num' is given in the X array}
        IF (NUM=k) STOP;

Step 5 - IF (COUNT=0) RETURN TO Step 3;
        X(R(COUNT)):=X(R(COUNT))-1;
        REF:=REF+A(R(COUNT));
        I:=R(COUNT)-1;
        COUNT:=COUNT-1;
        WHILE ((I>0) AND (M(REF,I)=0 OR X(I)=U(I))) DO
            I:=I-1;
        IF (I>0) THEN
        BEGIN
            COUNT:=COUNT+1;
            R(COUNT):=I;
            X(I):=X(I)+1;
            S:=R(COUNT);
            RETURN TO Step 4;
        END;
        RETURN TO Step 5;
```

References

1. M.R.GAREY, D.S.JOHNSON (1978) *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman.

2. C.A.KLUYVER, H.M.SALKIN (1975) The knapsack problem: a survey. *Naval Research Logistics Quarterly* **2**, 127-144.

3. E.HOROWITZ, S.SAHNI (1974) Computing partitions with applications to the knapsack problem. *Journal ACM* **21**, 277-292.

4. D.FAYARD, G.PLATEAU (1982) An algorithm for the solution of the 0-1 knapsack problem. *Computing* **28**, 269-287.

5. P.GILMORE, R.GOMORY (1966) The theory and computation of knapsack functions. *Operations Research* **14**, 1045-1074.

6. S.MARTELLO, P.TOTH (1979) The 0-1 knapsack problem. In *Combinatorial Optimization* (N.CHRISTOFIDES, A.MINGOZZI, P.TOTH, SANDI, Eds), Wiley, New York.

7. J.SHAPIRO, H.WAGNER (1967) A finite renewal algorithm for the knapsack and turnpike models. *Operations Research* **15**, 319-341.

8. E.LAWLER (1979) Fast approximations algorithms for knapsack problems. *Math. Oper. Res.* **4**, 339-356.

9. S.MARTELLO, P.TOTH *Knapsack Problems - Algorithms and Computer Implementations*. Wiley, 1990.

10. B.FAALAND (1973) Solution of the Value-Independent Knapsack Problem by Partitioning. *Operations Research* **21**, 332-337.

11. A.I.HINXMAN (1980) The trim loss and assortment problems: a survey. *EJOR* **5**, 8-18.

12. P.GILMORE, R.E.GOMORY (1963) A linear programming approach to the cutting stock problem. *Op.Res.* **11**, 863-888.

13. H.DYCKHOFF, D.ABEL, T.GAL.,H.J.KRUSE (1985) Trim loss and related problems. *OMEGA* **13**, 59-72.

14. H.H.YANASSE, N.Y.SOMA (1987) A new enumeration scheme for the knapsack problem. *Discrete Applied Mathematics* **18**, 235-245.

PRESIDÊNCIA DA REPÚBLICA
SECRETARIA DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

AUTORIZAÇÃO PARA PUBLICAÇÃO

MFN6087

TÍTULO

FINDING SOLUTIONS TO VALUE INDEPENDENT KNAPSACK PROBLEMS.

AUTOR

Horacio H. Yanasse; Nei Y. Soma

TRADUTOR

EDITOR

ORIGEM	PROJETO	SÉRIE	Nº DE PÁGINAS	Nº DE FOTOS	Nº DE MAPAS
LAC			28		

TIPO

☐ RPQ ☐ PRE ☐ NTC ☐ PRP ☐ MAN ☐ PUD ☒ TAE ☐ ___

DIVULGAÇÃO

☒ EXTERNA ☐ INTERNA ☐ RESERVADA ☐ LISTA DE DISTRIBUIÇÃO ANEXA

PERIÓDICO/EVENTO

CONVÊNIO

AUTORIZAÇÃO PRELIMINAR

___/___/___ ASSINATURA

REVISÃO TÉCNICA

☐ SOLICITADA ☐ DISPENSADA ASSINATURA

RECEBIDA ___/___/___ DEVOLVIDA ___/___/___ ASSINATURA DO REVISOR

REVISÃO DE LINGUAGEM

☐ SOLICITADA ☐ DISPENSADA ASSINATURA

Nº ___

RECEBIDA ___/___/___ DEVOLVIDA ___/___/___ ASSINATURA DO REVISOR

PROCESSAMENTO/DATILOGRAFIA

RECEBIDA ___/___/___ DEVOLVIDA ___/___/___ ASSINATURA

REVISÃO TIPOGRÁFICA

RECEBIDA ___/___/___ DEVOLVIDA ___/___/___ ASSINATURA

AUTORIZAÇÃO FINAL

17/04/92 Paulo Renato de Morai.
Chefe
Assinatura Tecnologias Associadas - C...
Substituto

PALAVRAS-CHAVE

knapsack, subset sum, linear diophantine equation, enumeration scheme.