Computing 27, 93—112 (1981)

Computing

Heuristic Algorithms for the Multiple Knapsack Problem

S. Martello and **P. Toth,** Bologna

Received November 10, 1980

Abstract — Zusammenfassung

Heuristic Algorithms for the Multiple Knapsack Problem. Given a set of items, each having a profit and a weight, and a set of knapsacks, each having a capacity, we consider the problem of inserting items into the knapsacks in such a way that the subset of inserted items has the maximum total profit without the total weight in each knapsack exceeding its capacity.

The best algorithms for the exact solution of this problem can be applied, with acceptable running times, to cases with a maximum of 200 items and 4 knapsacks, but real world applications (such as, for example, cutting stock and loading problems), often involving a greater number of variables, call for the use of heuristics.

This paper presents methods for finding suboptimal solutions to the Multiple Knapsack Problem. An extensive computational experience was carried out both on small-size and large-size randomly generated problems; the results indicate that the proposed algorithms have a satisfactory behaviour with regard both to running times and quality of the solutions found.

A FORTRAN IV implementation of the algorithms is given.

Heuristische Algorithmen zur Packung von mehreren Rucksäcken. Vorhanden ist eine Menge von Gegenständen mit gegebenem Gewicht und gegebenem Wert, ferner eine Menge von Rucksäcken mit gegebener Tragfähigkeit. Die Gegenstände sollen so in die Rucksäcke gepackt werden, daß die Tragfähigkeit der einzelnen Rucksäcke nicht überschritten wird und der Gesamtwert der verwendeten Gegenstände möglichst groß ist.

Die besten bekannten Algorithmen liefern mit vertretbarem Rechenaufwand Lösungen für höchstens 200 Gegenstände und 4 Rucksäcke. Aber praktische Anwendungen wie z. B. Teilungs-, Lagerhaltungs- und Ladeaufgaben benötigen oft eine größere Zahl von Variablen und verlangen daher die Verwendung von Heuristik.

Der vorliegende Artikel erläutert Methoden, mit deren Hilfe sich suboptimale Lösungen des Mehr-Rucksack-Problems finden lassen. Numerische Experimente mit Problemen verschiedener Größe zeigen, daß sich die verschiedenen Algorithmen zufriedenstellend verhalten, sowohl hinsichtlich der Laufzeiten als auch bezüglich der Güte der gefundenen Lösungen.

Eine FORTRAN-IV-Version der Algorithmen ist beigefügt.

1. Introduction

Given the sets $N=\{1, 2, \ldots, n\}$, $M=\{1, 2, \ldots, m\}$ and the vectors (p_j), (w_j), (c_i), we define the *Zero-One Multiple Knapsack Problem* as

$$\max \sum_{i \in M} \sum_{j \in N} p_j x_{i,j} \tag{1}$$

0010-485X/81/0027/0093/$ 04.00

$$(P) \qquad \text{subject to} \quad \sum_{j \in N} w_j x_{i,j} \leq c_i \qquad \text{for all } i \in M; \tag{2}$$

$$\sum_{i \in M} x_{i,j} \leq 1 \qquad \text{for all } j \in N; \tag{3}$$

$$x_{i,j} \in \{0, 1\} \qquad \text{for all } i \in M, j \in N. \tag{4}$$

The problem can be viewed as that of taking n items, each having a profit p_j and a weight w_j, and selecting those to be inserted into m boxes (knapsacks) of capacities c_i, so that the total profit of the inserted items is maximum (1), the total weight inserted in each knapsack does not exceed the corresponding capacity (2), while each item is either inserted in a knapsack or rejected (3, 4). The problem may represent many industrial situations such as the loading of m ships with n containers, the loading of n tanks with m liquids that cannot be mixed or the cutting of m unidimensional items into n pieces of assigned lengths.

We will assume, with no loss of generality, that

$$p_j, w_j, c_i > 0 \text{ and integers};$$

$$\min_j \{w_j\} \leq \min_i \{c_i\};$$

$$\max_j \{w_j\} \leq \max_i \{c_i\};$$

$$\sum_{j \in N} w_j > \max_i \{c_i\}.$$

Furthermore, we will assume that the items are arranged so that

$$p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n.$$

The problem represents a generalization of the well-known *Zero-One Single Knapsack Problem* (P_1), where only one knapsack is available for the n items.

An upper bound for the value of (P) can be obtained by solving the *Surrogate Relaxation* given by the single knapsack problem:

$$(P_S) \qquad \max \sum_{j \in N} p_j z_j$$

$$\text{subject to} \quad \sum_{j \in N} w_j z_j \leq \sum_{i \in M} c_i;$$

$$z_j \in \{0, 1\} \qquad \text{for all } j \in N.$$

Algorithms for the exact solution of the Zero-One Multiple Knapsack Problem have been presented by Hung and Fisk [2] and by Martello and Toth [4, 5]; this paper studies algorithms for finding an approximate solution to the problem.

The need for heuristics is justified by the computational complexity of the problem. It is in fact known that (P_1) is an NP-complete problem; since for any instance of (P_1) an instance of (P) can be constructed in polynomial time (by simply setting $m = 1$), which has a solution if and only if (P_1) has a solution,

(*P*) too is *NP*-complete. There is no proof but strong suspicion that no *NP*-complete problem can be solved by a polynomial-time algorithm, so the worst cases of these problems are computationally "intractable" and heuristic algorithms are useful for their solution.

To the authors' knowledge, the only heuristic algorithm for (*P*) that has been proposed is Fisk and Hung's [1]: this method exactly solves (P_S) (let the solution be $Z=\{j \mid z_j=1\}$) and then tries to insert the items of Z in the knapsacks; when an item cannot be inserted in any knapsack, for each pair of knapsacks it attempts exchanges between items (one for one, then two for one, then one for two) until an exchange is found which fully utilizes the available space in one of the knapsacks. If all the items of Z are inserted, an optimal solution is found; otherwise, the current (suboptimal) feasible solution can be improved by inserting in the knapsacks as many items of $N-Z$ as possible. In the worst case Fisk-Hung's algorithm requires a non-polynomial running time, since it needs exact solution of an *NP*-complete problem.

Our purpose is to give heuristic methods useful for those multiple knapsack problems, often arising in real world applications, which cannot be solved by means of non-polynomial algorithms because of the enormous amount of computation needed (as will be shown in Section 2); consequently, the procedures we propose, which are all polynomial in the problem size $m+n$, have been designed so as always to guarantee acceptable running times (we present computational results up to 100 knapsacks and 1000 items); since the theoretical analysis of worst-case bounds to the errors often gives results far from the actual performance of heuristics, we evaluated our approximations through extensive computational experiments on different types of randomly generated data sets reflecting most of the real-world situations. The results indicate that the proposed methods have a satisfactory behaviour with regard both to running times and quality of the solutions found.

Three kinds of procedures are developed: to find an initial feasible solution (Section 3), to improve on a feasible solution (Section 4), to rearrange a feasible solution (Section 5). These procedures are evaluated through computational experiments on small-size problems (i. e. problems having up to 200 items and up to 4 knapsacks). Computational results on large-size problems are given in Section 6. The FORTRAN codes are presented in Section 7.

We will always assume that the knapsacks are initially arranged so that

$$c_1 \leq c_2 \leq \ldots \leq c_m.$$

2. Computational Behaviour of Exact Algorithms for Knapsack Problems

We considered uniformly random data sets of different categories, corresponding to different distributions utilized for generating items and knapsacks.

Two distributions were used for the items:

a) $10 \leq w_j \leq 100$, $10 \leq p_j \leq 100$ for all $j \in N$ (*Uncorrelated items*);

b) $10 \leq w_j \leq 100$, $p_j = w_j + 10$ for all $j \in N$ (*Correlated items*).

Two distributions were used for the knapsacks:

c) $\left[0.4 \sum_{j \in N} w_j/m\right]^* \leq c_i \leq \left[0.6 \sum_{j \in N} w_j/m\right]$ for $i = 1, \ldots, m-1$ (*Similar knapsacks*);

d) $0 \leq c_1 \leq \left[0.5 \sum_{j \in N} w_j\right]$,

$0 \leq c_i \leq \left[0.5 \sum_{j \in N} w_j\right] - \sum_{u<i} c_u$ for $i = 2, \ldots, m-1$ (*Dissimilar knapsacks*).

Both for c) and for d) the m-th knapsack capacity was chosen such that $\sum_{i \in M} c_i = \left[0.5 \sum_{j \in N} w_j\right]$; if $c_i < \min_{j \in N} \{w_j\}$ for some i or $\max_{i \in M} \{c_i\} < \max_{j \in N} \{w_j\}$, a new set of knapsack capacities was generated.

For each category, different pairs m, n were considered and, for each pair, 30 problems were generated.

For the optimal solution of the single knapsack problems, required by the methods in [1, 2, 4, 5], the branch and bound algorithm of Martello and Toth [3] was employed.

All the algorithms were coded in FORTRAN IV and run on a CDC-6600; the entries in the tables give the average running times expressed in milliseconds.

Let us compare (Table 1) exact algorithms for the single and for the multiple knapsack problem on uncorrelated items and similar knapsacks.

Table 1. *Exact algorithms. (Uncorrelated items — similar knapsacks — CDC-6600 milliseconds)*

		$n=25$	$n=50$	$n=100$	$n=200$
Single K.P. (Algorithm in [3])	$m=1$	4	6	12	21
Multiple K.P. (Algorithm in [5])	$m=2$	106	237	636	3344
	$m=3$	543	377	802	3563
	$m=4$	1649	4751	1490	6818

The single knapsack problem does not show its complexity: its average times grow almost linearly with the value of n. On the contrary, the multiple knapsack problem fully shows its exponential growing rate and appears to be intractable for $m > 4$ or for n greater than a few hundred. This difference between the two cases is given by the enormous (exponential) number of single knapsack problem solutions that all the algorithms for the multiple knapsack problem involve.

If we now consider the case of correlated items, where the single knapsack problem too shows its complexity (for capacity $= \left[0.5 \sum_{j \in N} w_j\right]$, the algorithm in [3] required 36 milliseconds for $n = 25$, 2700 milliseconds for $n = 50$), it is clear that

* $[a]$ = largest integer not greater than a.

for such data no exact algorithm for the multiple knapsack problem could be used and that even the heuristic method of Fisk and Hung [1] is impractical because of the need to solve the surrogate relaxation of the problem exactly.

The test problems described in this section are used in the following to evaluate the heuristic procedures we present. In the tables of the next sections, each entry gives two values: the average running time (expressed in CDC-6600 milliseconds and relating to FORTRAN IV codes) and, in brackets, an upper bound on the percentage error, that is 100 ε, where ε is computed as follows: let

V^* = value of the heuristic solution found for (P);
V_S = value of the optimal solution of (P_S);
$\bar{V}_S$ = upper bound on V_S, computed as in [3];

for uncorrelated items we set

$$\varepsilon = (V_S - V^*)/V_S,$$

while for correlated items, for which the computation of V_S is too hard, we set

$$\varepsilon = (\bar{V}_S - V^*)/\bar{V}_S.$$

3. Algorithms for Finding an Initial Feasible Solution

Some of the methods we present require, as a subproblem, the heuristic solution of a single knapsack problem; this was always obtained through the following well-known *Greedy Algorithm*:

procedure GREEDY (A, c, G)
    input: $A = \{j \mid \text{item } j \text{ is available}\}$ $(A \subseteq N)$;
        c = available capacity.
    output: $G = \{j \mid \text{item } j \text{ is inserted in the knapsack}\}$ $(G \subseteq A)$.
Set $G = \emptyset$.
For increasing $j \in A$ do
    if $c\, p_j/w_j < 1$, return;
    if $w_j \leq c$, set $c = c - w_j$, $G = G \cup \{j\}$;
repeat.
Return.

In the worst case, this procedure requires $|A|$ iterations, that is the number of operations is $O(n)$.

The following algorithms can be used to obtain an initial feasible solution G_i $(i = 1, \ldots, m)$ to (P).

Algorithm MK 1:

Set $R = N$.
For increasing $i \in M$ do
    apply GREEDY (R, c_i, G_i);
    set $R = R - G_i$;
repeat.
Stop.

MK1 applies GREEDY m times, that is it requires, in the worst case, $O(m\,n)$ operations.

Algorithm MK2:

Set $G_i=\emptyset$ for all $i\in M$, $i=1$.
For increasing $j\in N$ do
  set $k=1$;
  while $w_j>c_i$ and $k\leq m$ do
    set $k=k+1$, $i=i+1$; if $i>m$, set $i=1$;
  repeat;
  if $w_j\leq c_i$, set $c_i=c_i-w_j$, $G_i=G_i\cup\{j\}$, $i=i+1$; if $i>m$, set $i=1$;
repeat.
Stop.

MK2 inserts the first item in the first knapsack, the second item in the second knapsack, and so on cyclically; when an item cannot be inserted in the required knapsack, the next knapsack is tried, and so on; items which cannot be inserted in any knapsack are rejected. In the worst case, m iterations are to be performed for each item and the algorithm requires $O(m\,n)$ iterations.

Algorithm MK3:

For increasing $i\in M$ do
  set $\bar{c}=c_i$;
  apply GREEDY $(N, \bar{c}, G_i)$;
repeat.

For increasing $j\in N$ do:
  set $B=\{i\mid j\in G_i\}$;
  if $|B|>1$, then set $A=\{q\mid q\in N,\ q>j\}$, $\Delta^*=-\infty$, and for increasing
    $i\in B$ do:
      set $\tilde{G}=G_i$, $\bar{c}=c_i$;
      apply GREEDY $(A, \bar{c}, \bar{G})$;
      set $G_i=\bar{G}\cup\{q\mid q\in G_i,\ q<j\}$;
      if $\Delta^*<\Delta=\sum_{q\in\bar{G}} p_q-\sum_{q\in G_i} p_q$, set $i^*=i$, $\Delta^*=\Delta$, $G^*=\tilde{G}$;
    repeat;
    set $G_{i^*}=G^*$, $c_{i^*}=c_{i^*}-w_j$;
  else, if $|B|=1$ (say $B=\{i^*\}$), set $c_{i^*}=c_{i^*}-w_j$;
repeat.
Stop.

MK3 starts by finding m greedy solutions for the m knapsacks, but in this case the items of each G_i are not excluded from N, so each item will generally appear in a number of knapsacks; in such cases the item is assigned to the knapsack for which the loss of profit corresponding to the exclusion of the item would be the greatest and the current greedy solutions of the remaining knapsacks are updated. In the worst case, MK3 requires that, for each item, m greedy solutions be computed, so the algorithm's complexity is $O(m\,n^2)$.

In the innermost loop an increase in efficiency can be obtained by substituting the application of GREEDY $(A, \bar{c}, \bar{G})$ by a faster greedy procedure which takes into account the obvious fact that the first s items in $\tilde{G} \cap A$, where

$$s = \max \left\{ q \mid \sum_{t=j}^{j+q} w_t \leq c_i \right\},$$

must be in $\bar{G}$ too.

It should be noted that in MK1 and MK2 the G_i's are disjoint, so we need just a vector ($g_j = i$ if item j is inserted in knapsack i, for $j = 1, \ldots, n$) in order to store all the information in G_i $(i = 1, \ldots, m)$.

Table 2 gives the computational performace of the three MK algorithms for the data set b)–d), that is (see Section 2) correlated items and dissimilar knapsacks; the other possible data sets (b)–c), a)–d) and a)–c)) showed about the same algorithm behaviour. MK3 had in general the best percentage errors but also the worst running times.

Table 2. *Correlated items — dissimilar knapsacks. CDC-6600 milliseconds (percentage error)*

m	n	MK1	MK2	MK3
2	25	1 (6.13)	1 (7.08)	5 (4.64)
	50	1 (3.15)	2 (3.88)	9 (2.08)
	100	2 (1.73)	4 (2.11)	18 (1.34)
	200	5 (0.88)	7 (1.08)	41 (0.57)
3	25	1 (8.00)	1 (9.65)	6 (6.50)
	50	2 (4.43)	2 (5.03)	13 (3.06)
	100	4 (2.10)	4 (2.57)	24 (1.39)
	200	8 (1.06)	8 (1.22)	51 (0.70)
4	25	2 (9.61)	1 (12.49)	6 (9.93)
	50	3 (5.51)	2 (5.49)	14 (4.62)
	100	6 (2.73)	5 (2.88)	28 (2.19)
	200	11 (1.08)	10 (1.43)	60 (1.20)

4. Algorithms for Improving on a Feasible Solution

Assume a feasible solution G_i $(i = 1, \ldots, m)$ is known and define:

$S = \bigcup_{i=1}^{m} G_i$;

$R = N - S$;

v_i = total profit currently inserted in knapsack i;

c_i = current remaining capacity of knapsack i;

g_j = knapsack where item j is currently inserted;

where S and R (subsets of N) are assumed to be in nonincreasing order of p_j/w_j; in the algorithms of this Section we will suppose that S, R, v_i, c_i and g_j are updated whenever some G_i is updated.

The following algorithms can be used to improve on a feasible solution.

Algorithm I1:

For increasing $j_1 \in \{j \mid j \in S,\ c_{g_j} + \max_{i \neq g_j} \{c_i\} \geq \min_{r \in R} \{w_r\}\}$ do

for increasing $j_2 \in \{j \mid j \in S, j > j_1, g_j \neq g_{j_1}, c_{g_j} + c_{g_{j_1}} \geq \min_{r \in R} \{w_r\}\}$ do

let $w_{j_u} = \max \{w_{j_1}, w_{j_2}\}$, $w_{j_q} = \min \{w_{j_1}, w_{j_2}\}$, $i_u = g_{j_u}$, $i_q = g_{j_q}$;
set $\delta = w_{j_u} - w_{j_q}$;
if $\delta \leq c_{i_q}$ and $c_{i_u} + \delta \geq \min_{r \in R} \{w_r\}$, set $p_t = \max \{p_r \mid r \in R,\ w_r \leq c_{i_u} + \delta\}$,
$G_{i_u} = (G_{i_u} - \{j_u\}) \cup \{j_q, t\}$, $G_{i_q} = (G_{i_q} - \{j_q\}) \cup \{j_u\}$;

repeat;
repeat.
Stop.

I1 considers all pairs of items in S and, if possible, interchanges them whenever doing so allows the insertion of an item from R into one of the knapsacks; the algorithm's complexity is $O(n^2)$ (the search for p_t is executed at most $|R|$ times). It should be noted that it is computationally efficient to stop the execution as soon as $\max_{i \neq l} \{c_i + c_l\} < \min_{r \in R} \{w_r\}$.

Algorithm I2:

Set $\bar{v} = 0$, $k = 1$, $l =$ maximum number of iterations.

While $\bar{v} < \sum_{i \in M} v_i$ and $k \leq l$ do

set $\bar{v} = \sum_{i \in M} v_i$, $k = k + 1$;
for decreasing $j \in S$ do
let i be the knapsack such that $j \in G_i$ and set $\bar{c} = c_i + w_j$;
apply GREEDY $(R, \bar{c}, L)$;
if $\sum_{q \in L} p_q > p_j$, set $G_i = (G_i - \{j\}) \cup L$;
repeat;
repeat.
Stop.

This algorithm tries to exclude in turn each item currently in S and to replace it with one or more items from R so that the total profit is increased; the execution stops when no further replacement is possible or when l complete iterations have been performed. If l is a linear function of n, the complexity of I2 is $O(n^3)$; we always assumed $l = n$.

The separate and combined effects of I1 and I2 on the solutions found by the MK algorithms were experimentally tested: I2 produced better improvements than I1 but required higher times; the sequence I1+I2 required about the same times as I2 and generally gave a better improvement; so, in what follows, we will always apply I1 followed by I2 (the sequence I2+I1 generally gave worse results).

The first three columns of Table 3 show, for the same data set as Table 2, the effect of the sequence I1+I2 applied to the solutions of algorithms MK1, MK2 and MK3.

It should be noted that MK2, which had bad initial solutions, obtained great improvements, leading it to achieve in general good final solutions; this behaviour is analyzed in the next Section.

Table 3. *Correlated items — dissimilar knapsacks. CDC-6600 milliseconds (percentage error)*

m	n	MK1+I1+I2	MK2+I1+I2	MK3+I1+I2	MK1+A+I1+I2	MK3+A+I1+I2
2	25	4 (1.64)	4 (1.61)	6 (1.74)	5 (1.21)	8 (1.18)
	50	9 (0.59)	9 (0.46)	14 (0.62)	11 (0.34)	18 (0.34)
	100	42 (0.34)	47 (0.26)	51 (0.35)	52 (0.20)	64 (0.20)
	200	97 (0.15)	115 (0.15)	113 (0.15)	127 (0.08)	158 (0.08)
3	25	5 (2.95)	5 (2.75)	8 (2.24)	6 (1.91)	10 (1.97)
	50	13 (1.63)	13 (1.34)	16 (0.85)	14 (0.50)	22 (0.50)
	100	31 (0.62)	41 (0.46)	38 (0.28)	35 (0.18)	53 (0.18)
	200	103 (0.28)	183 (0.23)	120 (0.16)	133 (0.08)	177 (0.08)
4	25	6 (4.37)	5 (4.08)	9 (2.54)	7 (2.48)	12 (2.12)
	50	15 (1.88)	16 (1.40)	20 (0.78)	20 (0.70)	26 (0.62)
	100	34 (0.76)	49 (0.67)	46 (0.35)	38 (0.21)	55 (0.23)
	200	120 (0.33)	193 (0.27)	104 (0.14)	124 (0.08)	166 (0.08)

5. Algorithm for Rearranging a Feasible Solution

The particular performance of I1 and I2 when applied to the solutions found by MK2 can be explained by comparing the structure of these solutions with that of the MK1 solutions.

Consider what is the situation expected when, for each knapsack, the first item is found which cannot be inserted. In the MK1 solutions, when this happens for the first knapsacks, generally several items of smaller weight are available, so the algorithm can fill such knapsacks well, whereas when this happens for the last knapsacks, only items of considerable weight are available so the knapsacks are not well filled; in the MK2 solutions the last situation occurs for all the knapsacks. Hence the worse performance of MK2 but also the different structure of the solutions, viz. each knapsack filled with items of similar profit per unit weight in the MK1 solutions, with items of dissimilar profit per unit weight in the MK2 solutions: this explains why the improving algorithms, which are based on exchanges among the items, work very well on the MK2 solutions.

The following algorithm can be used for rearranging the feasible solutions found by the MK algorithms in order to obtain a structure similar to that of the MK2 solutions.

Algorithm A:

Set c_i $(i=1, \ldots, m)$ to its original value.
Set $G_i=\emptyset$ for all $i \in M$, $i=1$.
For decreasing $j \in S$ do
    set $k=1$;
    while $w_j > c_i$ and $k \leq m$ do
        set $k=k+1$, $i=i+1$; if $i>m$, set $i=1$;
    repeat;
    if $w_j \leq c_i$, set $c_i=c_i-w_j$, $G_i=G_i \cup \{j\}$, $i=i+1$; if $i>m$, set $i=1$;
repeat.
For increasing $i \in M$ do
    apply GREEDY (R, c_i, L);
    set $R=R-L$, $G_i=G_i \cup L$;
repeat.
Stop.

The complexity of A is obviously the same as MK2, i.e. $O(m\,n)$ operations. The last two columns of Table 3 show the results obtained by using A before the sequence I1+I2: A required an extra computational effort, but always produced a great reduction in the percentage error. The results given by MK2+I1+I2 were always worse, both for running time and percentage error, than those given by MK1+A+I1+I2, so MK2 will not be considered any further.

6. Computational Results

The algorithms obtained in the previous section (MK1+A+I1+I2 and MK3+A+I1+I2) were compared with the algorithm of Fisk and Hung [1] (here referred to as FH) on the small-size data sets considered up to now, that is on problems having up to 200 items and up to 4 knapsacks. The following results were obtained:

i) Uncorrelated items-Similar knapsacks: MK1+A+I1+I2 obtained satisfactory approximations and required low running times; MK3+A+I1+I2 gave slightly better approximations but required a computational effort about five times higher; FH generally had the best approximations with intermediate times; the difference between the approximations of FH and those of the other algorithms seem to decrease when m grows.

ii) Uncorrelated items-Dissimilar knapsacks: MK1+A+I1+I2 and MK3+A+I1+I2 gave about the same results as in case i); FH required running times slightly higher than those of MK1+A+I1+I2 and had the best approximations for $m=2, 3$ but the worst approximations for $m=4$.

iii) Correlated items-Similar knapsacks: MK1+A+I1+I2 and MK3+A+I1+I2 obtained about the same approximations but the latter required 50% higher running times; FH could not solve to within the time limit of 250 seconds data sets having more than 50 items; for $n=25, 50$, it generally gave

good approximations but required enormous running times (about 100 times those of the other algorithms for $n=50$).

iv) Correlated items-Dissimilar knapsacks: MK1+A+I1+I2 and MK3+A+I1+I2 gave about the same results as in case iii) (see also Table 3); FH had about the same behaviour as in case iii), as far as the running times are concerned, but generally gave the worst approximations.

Coming to large-size data sets, that is to problems having up to 1000 items and up to 100 knapsacks, we followed the information given by the small-size problems and tested algorithms MK1+A+I1+I2 and FH for uncorrelated items, and algorithm MK1+A+I1+I2 for correlated items; because of the high running times, we also give the intermediate results corresponding to MK1 and to MK1+A+I1. In order to avoid anomalous problems, we always considered data sets such that $n/m \geq 4$.

Table 4 shows that, for uncorrelated items and dissimilar knapsacks, FH obtained the best approximations but required the highest running times, while MK1+A+I1+I2 gave average approximations with low running times.

Table 4. *Uncorrelated items — similar knapsacks. CDC-6600 milliseconds (percentage error)*

m	n	MK1	MK1+A+I1	MK1+A+I1+I2	FH
10	50	7 (4.47)	12 (3.97)	16 (3.37)	20 (2.70)
	100	12 (1.73)	18 (1.29)	24 (1.06)	31 (0.25)
	200	21 (0.66)	32 (0.41)	43 (0.37)	67 (0.10)
	500	49 (0.19)	75 (0.13)	102 (0.11)	368 (0.04)
	1000	95 (0.09)	139 (0.05)	199 (0.03)	1524 (0.02)
20	100	25 (3.72)	37 (2.99)	45 (2.66)	52 (1.65)
	200	44 (1.30)	62 (0.97)	78 (0.89)	79 (0.16)
	500	104 (0.43)	130 (0.29)	186 (0.22)	256 (0.06)
	1000	195 (0.18)	247 (0.12)	347 (0.08)	1236 (0.04)
50	200	114 (4.33)	151 (4.29)	206 (3.52)	227 (1.38)
	500	258 (1.14)	323 (0.84)	408 (0.71)	429 (0.23)
	1000	481 (0.44)	580 (0.31)	755 (0.24)	Core Memory
100	500	521 (2.46)	747 (2.16)	917 (1.81)	Core Memory
	1000	1010 (1.02)	1194 (0.76)	1522 (0.62)	Core Memory

Table 5 refers to uncorrelated items and dissimilar knapsacks: FH obtained the worst approximations; MK1 had clearly better approximations and generally lower running times; A, I1 and I2 required a strong extra computational effort, giving only small improvements. Both in Table 4 and in Table 5 FH, which requires an $(m \times n)$ matrix, was not applied to data sets such that $m\,n \geq 50000$.

Table 5. *Uncorrelated items — dissimilar knapsacks. CDC-6600 milliseconds (percentage error)*

m	n	MK1	MK1+A+I1	MK1+A+I1+I2	FH
10	50	8 (3.06)	17 (3.23)	20 (2.62)	15 (8.40)
	100	14 (1.13)	32 (0.85)	36 (0.81)	37 (2.41)
	200	25 (0.39)	64 (0.30)	73 (0.26)	84 (0.84)
	500	55 (0.12)	148 (0.08)	171 (0.07)	229 (0.30)
	1000	114 (0.07)	598 (0.04)	648 (0.03)	536 (0.16)
20	100	34 (3.14)	62 (3.03)	67 (2.86)	37 (11.63)
	200	65 (1.30)	131 (1.37)	141 (1.20)	72 (5.12)
	500	138 (0.41)	327 (0.37)	352 (0.34)	204 (1.84)
	1000	230 (0.16)	1390 (0.13)	1443 (0.11)	947 (0.74)
50	200	192 (4.43)	297 (4.38)	307 (4.31)	135 (16.21)
	500	470 (1.63)	771 (1.59)	795 (1.55)	372 (7.18)
	1000	881 (0.73)	2051 (0.71)	2113 (0.69)	Core Memory
100	500	994 (3.61)	1651 (3.55)	1677 (3.53)	Core Memory
	1000	1927 (1.74)	3386 (1.72)	3437 (1.70)	Core Memory

Table 6 (correlated items – similar knapsacks) shows that using A+I1+I2 requires a strong extra computational effort but gives a strong reduction in the percentage errors; using only A+I1 requires a small extra computational effort and gives a good improvement in approximation.

Table 6. *Correlated items — similar knapsacks. CDC-6600 milliseconds (percentage error)*

m	n	MK1	MK1+A+I1	MK1+A+I1+I2
10	50	6 (17.42)	13 (6.81)	28 (3.40)
	100	12 (7.37)	20 (2.99)	66 (0.65)
	200	24 (3.95)	37 (1.03)	225 (0.21)
	500	58 (1.69)	90 (0.40)	990 (0.04)
	1000	118 (0.81)	168 (0.18)	2681 (0.01)
20	100	27 (17.69)	47 (6.48)	127 (2.78)
	200	52 (7.51)	76 (2.65)	366 (0.59)
	500	123 (3.11)	157 (0.78)	812 (0.04)
	1000	243 (1.58)	308 (0.31)	3607 (0.02)
50	200	137 (13.33)	209 (9.71)	950 (4.76)
	500	319 (7.65)	466 (2.85)	3139 (0.30)
	1000	631 (3.94)	809 (1.10)	6162 (0.05)
100	500	671 (17.42)	1469 (5.07)	5391 (2.75)
	1000	1272 (7.65)	2023 (2.97)	16847 (0.22)

Table 7 shows analogous behaviour for correlated items and dissimilar knapsacks, but in this case both the extra computational effort and the improvements given by A, I1 and I2 are smaller. Both in Table 6 and in Table 7 FH is not considered since the required exact solutions of (P_S) is computationally intractable for correlated items.

Table 7. *Correlated items — dissimilar knapsacks. CDC-6600milliseconds (percentage error)*

m	n	MK1	MK1+A+I1	MK1+A+I1+I2
10	50	8 (7.93)	15 (5.32)	24 (3.30)
	100	16 (3.85)	33 (2.16)	77 (0.91)
	200	32 (1.80)	92 (0.90)	252 (0.27)
	500	83 (0.75)	139 (0.24)	736 (0.05)
	1000	160 (0.42)	520 (0.11)	2487 (0.02)
20	100	36 (6.89)	69 (5.67)	105 (4.64)
	200	69 (3.16)	134 (2.19)	242 (1.67)
	500	156 (1.18)	346 (0.65)	886 (0.45)
	1000	276 (0.48)	1499 (0.24)	4402 (0.14)
50	200	198 (8.71)	316 (7.51)	421 (7.06)
	500	481 (3.14)	869 (2.69)	1664 (2.49)
	1000	898 (1.44)	1699 (1.22)	3784 (1.12)
100	500	1033 (6.53)	1621 (6.05)	2334 (5.84)
	1000	1955 (3.02)	3608 (2.85)	6133 (2.76)

For each entry of Tables 4, 5, 6, 7, computation of the ratios ϱ_t = (maximum running time)/(average running time) and ϱ_ε = (maximum percentage error)/(average percentage error) gave the following results:

MK1: $\varrho_t \leq 2$ and $\varrho_\varepsilon \leq 2$.

MK1+A+I1: for similar knapsacks, $\varrho_t \leq 3$ and $\varrho_\varepsilon \leq 2$; for dissimilar knapsacks, generally $\varrho_t \leq 4$ and $\varrho_\varepsilon \leq 2$ (with a few entries having maximum ratios $\varrho_t \cong 10$ or $\varrho_\varepsilon \cong 3$).

MK1+A+I1+I2: ϱ_t as MK1+A+I1; generally $\varrho_\varepsilon \leq 2$ (maximum $\varrho_\varepsilon \cong 4$).

FH: for similar knapsacks, $\varrho_t \leq 6$ and generally $\varrho_\varepsilon \leq 7$ (maximum $\varrho_\varepsilon \cong 20$); for dissimilar knapsacks, generally $\varrho_t \leq 7$ and $\varrho_\varepsilon \leq 2$ (maximum $\varrho_t \cong 10$ or $\varrho_\varepsilon \cong 4$).

From Tables 4, 5, 6, 7 we can conclude that for the solution of real world applications involving large-size multiple knapsack problems it is impossible to indicate one algorithm as the best for all cases, but, according to the kind of problems to be solved, the best methods are:

a) Uncorrelated items: for similar knapsacks, FH up to a few hundred items, then MK1+A+I1+I2; for dissimilar knapsacks, MK1.

b) Correlated items: for both similar and dissimilar knapsacks, MK1+A+I1 or, if tight approximations are needed, MK1+A+I1+I2; it should be noted that, in general, the improvement given by I2 strictly depends on the computing time required, so intermediate results can be obtained by imposing a maximum number of iterations $l < n$.

7. FORTRAN Subroutines

We present the FORTRAN codes of algorithms MK1, MK3, A, I1 and I2 of the previous Sections. Each algorithm is coded as a subroutine; in addition we give a subroutine (HMKP) to select the sequence of calls.

The package is completely self-contained and communication to it is solely made through the parameter list of HMKP. It is required that the items are already arranged in non-increasing order of the ratio p_j/w_j and the knapsacks in non-decreasing order of c_i.

As presently dimensioned, the size limitation is $n \leq 1000$ and $m \leq 100$ if MK1 is called, $n \leq 200$ and $m \leq 5$ if MK3 is called.

```
      SUBROUTINE HMKP(N,P,W,M,C,LM,LA,LI,VSTAR,Y,CR)
C     THIS SUBROUTINE HEURISTICALLY SOLVES THE 0-1 MULTIPLE KNAPSACK PROBLEM
C
C MAX VSTAR = P(1)*(X(1,1) + ... + X(M,1)) + ... + P(N)*(X(1,N) + ... + X(M,N))
C SUBJECT TO
C     W(1)*X(I,1) + ... + W(N)*X(I,N) NOT GREATER THAN C(I) (FOR I=1,...,M)
C     X(J,1) + ... + X(J,M) NOT GREATER THAN 1 (FOR J=1,...,N)
C     X(I,J) = 0 OR 1 (FOR I=1,...,M), (FOR J=1,...,N)
C
C     SUBROUTINE HMKP CAN CALL 5 SUBROUTINES, WHICH PERFORM ALGORITHMS TO FIND
C AN INITIAL FEASIBLE SOLUTION (MK1 OR MK3),TO REARRANGE A FEASIBLE SOLUTION (A)
C AND TO IMPROVE ON A FEASIBLE SOLUTION (I1 AND I2). THE USER SHOULD SELECT THE
C SEQUENCE OF CALLS THROUGH INPUT PARAMETERS LM, LA AND LI.
C
C     THE PACKAGE GIVEN BY SUBROUTINES HMKP, MK1, MK3, A, I1, I2 IS COMPLETELY
C SELF CONTAINED AND COMMUNICATION TO IT IS ACHIEVED SOLELY THROUGH THE
C PARAMETERS LIST OF HMKP. ALL THE PARAMETERS ARE INTEGER.
C
C     THE MEANING OF THE INPUT PARAMETERS
C N      = NUMBER OF ITEMS
C P(J)   = PROFIT OF THE J-TH ITEM (J=1,...,N)
C W(J)   = WEIGHT OF THE J-TH ITEM (J=1,...,N)
C M      = NUMBER OF KNAPSACKS
C C(I)   = CAPACITY OF THE I-TH KNAPSACK
C LM     = 1 TO PERFORM ALGORITHM MK1 (SUGGESTED FOR LARGE-SIZE PROBLEMS)
C        = 3 TO PERFORM ALGORITHM MK3 (SUGGESTED FOR SMALL-SIZE PROBLEMS)
C LA     = 0 NOT TO PERFORM ALGORITHM A
C        = 1 TO PERFORM ALGORITHM A (SUGGESTED IF I1 AND/OR I2 ARE PERFORMED)
C LI     = 0 TO PERFORM NEITHER ALGORITHM I1 NOR ALGORITHM I2
C        = 1 TO PERFORM ALGORITHM I1
C        = 2 TO PERFORM ALGORITHM I2
C        = 3 TO PERFORM ALGORITHM I1 AND THEN ALGORITHM I2
C
C     THE MEANING OF THE OUTPUT PARAMETERS
C VSTAR = VALUE OF THE SOLUTION FOUND
C Y(J)   = 0 IF ITEM J IS NOT IN THE SOLUTION FOUND (X(I,J) = 0 FOR I=1,...,M)
C        = II IF ITEM J IS IN THE SOLUTION FOUND (X(II,J) = 1)
C CR(I) = C(I) - (W(1)*X(I,1) + ... + W(N)*X(I,N)) (FOR I=1,...,M)
C
C     BEFORE HMKP IS CALLED, VECTORS P AND W NEED TO BE ARRANGED IN DECREASING
C ORDER OF THE RATIO P(J)/W(J), VECTOR C IN INCREASING ORDER OF C(I).
C
C     THE CALLING PROGRAM SHOULD CONTAIN THE FOLLOWING TYPE STATEMENT
C INTEGER P(N),W(N),C(M),Y(N),CR(M),VSTAR
C     LOCAL ARRAYS IN MK3 ARE CURRENTLY DIMENSIONED FOR PROBLEMS WITH M UP TO 5
C AND N UP TO 200, LOCAL ARRAYS IN I1 AND I2 FOR PROBLEMS WITH M UP TO 100 AND
C N UP TO 1000.
C
      INTEGER P(N),W(N),Y(N),C(M),CR(M),VSTAR
      IF ( LM .EQ. 3 ) GO TO 10
      CALL MK1(N,P,W,M,C,VSTAR,Y,CR)
      GO TO 20
   10 CALL MK3(N,P,W,M,C,VSTAR,Y,CR)
   20 IF ( LA .EQ. 1 ) CALL A(N,P,W,M,C,VSTAR,Y,CR)
      LL = LI + 1
      GO TO ( 60 , 30 , 50 , 40 ) , LL
   30 CALL I1(N,P,W,M,VSTAR,Y,CR)
      RETURN
   40 CALL I1(N,P,W,M,VSTAR,Y,CR)
   50 CALL I2(N,P,W,M,VSTAR,Y,CR)
   60 RETURN
      END

      SUBROUTINE MK1(N,P,W,M,C,VSTAR,Y,CR)
      INTEGER P(N),W(N),C(M),Y(N),CR(M),VSTAR
      INF = 999999999
      DO 10 I=1,M
        CR(I) = C(I)
```

```
   10 CONTINUE
      MP1 = M + 1
      P(N+1) = 0
      W(N+1) = INF
      J = 1
      I = 1
      VSTAR = 0
   20 IF ( W(J) .GT. CR(I) ) GO TO 30
      Y(J) = I
      CR(I) = CR(I) - W(J)
      VSTAR = VSTAR + P(J)
      J = J + 1
      GO TO 20
   30 JS = J
      Y(J) = MP1
      J = J + 1
      DO 40 JJ=J,N
        Y(JJ) = MP1
        IF ( W(JJ) .GT. CR(I) ) GO TO 40
        Y(JJ) = I
        CR(I) = CR(I) - W(JJ)
        VSTAR = VSTAR + P(JJ)
   40 CONTINUE
   50 IF ( I .LT. M ) GO TO 60
      GO TO 110
   60 I = I + 1
      DO 70 J=JS,N
        IF ( Y(J) .LE. M ) GO TO 70
        IF ( W(J) .GT. CR(I) ) GO TO 80
        Y(J) = I
        CR(I) = CR(I) - W(J)
        VSTAR = VSTAR + P(J)
   70 CONTINUE
      GO TO 110
   80 JS = J
      J = J + 1
   90 IF ( CR(I)*P(J)/W(J) .EQ. 0 ) GO TO 50
      IF ( Y(J) .LE. M ) GO TO 100
      IF ( W(J) .GT. CR(I) ) GO TO 100
      Y(J) = I
      CR(I) = CR(I) - W(J)
      VSTAR = VSTAR + P(J)
  100 J = J + 1
      GO TO 90
  110 CONTINUE
      RETURN
      END

      SUBROUTINE MK3(N,P,W,M,C,VSTAR,Y,CR)
      INTEGER P(N),W(N),C(M),Y(N),CR(M),VSTAR
      INTEGER S,D,ZCAP,VZCAP,CZCAP,VCAP,ZSTAR,VZSTAR,CZSTAR,Q
C     INTEGER A(M,N  ),V(M),Z(M),CZ(M),VZ(M),OZ(M),B(M),IFB(M),MINW(N+1)
      INTEGER A(5,200),V(5),Z(5),CZ(5),VZ(5),OZ(5),B(5),IFB(5),MINW(201)
      INF = 999999999
      VSTAR = 0
      JSTAR = 1
      P(N+1) = 0
      W(N+1) = INF
      MP1 = M + 1
      MINK = INF
      MINW(N+1) = MINK
      DO 20 J=1,N
        Y(J) = MP1
        KK = N + 1 - J
        IF ( W(KK) .LT. MINK ) MINK = W(KK)
        MINW(KK) = MINK
        DO 10 I=1,M
          A(I,J) = 0
   10   CONTINUE
   20 CONTINUE
      DO 30 I=1,M
        Z(I) = 1
        CZ(I) = C(I)
        VZ(I) = 0
        OZ(I) = 0
        B(I) = I
   30 CONTINUE
      IBOUND = 0
      KB = 0
      MB = M
   40 IF ( KB .EQ. MB ) GO TO 170
      KB = KB + 1
```

```
      I = B(KB)
      IF ( IBOUND .EQ. 0) GO TO 50
      ZCAP = Z(I)
      VZCAP = VZ(I)
      CZCAP = CZ(I)
      VCAP = V(I)
      IF ( S .GE. Z(I) ) GO TO 50
      VZ(I) = VZ(I) - P(S)
      CZ(I) = CZ(I) + W(S)
   50 J = Z(I)
      CR(I) = CZ(I)
      V(I) = VZ(I)
   60 IF ( CR(I) .LT. MINW(J) ) GO TO 70
      IF ( CR(I)*P(J)/W(J) .GE. 1 ) GO TO 80
   70 Z(I) = J
      CZ(I) = CR(I)
      VZ(I) = V(I)
      GO TO 140
   80 IF ( W(J) .GT. CR(I) ) GO TO 90
      CR(I) = CR(I) - W(J)
      V(I) = V(I) + P(J)
      A(I,J) = 1
      IOZ = J
      J = J + 1
      GO TO 60
   90 IF ( J .NE. JSTAR ) GO TO 100
      A(I,J) = 0
      J = J + 1
      GO TO 60
  100 Z(I) = J
      CZ(I) = CR(I)
      VZ(I) = V(I)
  110 IF ( CR(I) .LT. MINW(J) ) GO TO 140
      IF ( CR(I)*P(J)/W(J) .LT. 1 ) GO TO 140
      IF ( W(J) .GT. CR(I) ) GO TO 120
      CR(I) = CR(I) - W(J)
      V(I) = V(I) + P(J)
      A(I,J) = 1
      IOZ = J
      GO TO 130
  120 A(I,J) = 0
  130 J = J + 1
      GO TO 110
  140 JO = OZ(I)
      IF ( JO .LT. J ) GO TO 160
      DO 150 Q=J,JO
        A(I,Q) = 0
  150 CONTINUE
  160 OZ(I) = IOZ
      IF ( IBOUND .EQ. 0 ) GO TO 40
      IF ( VCAP - V(I) .LE. D ) GO TO 40
      D = VCAP - V(I)
      ISTAR = I
      ZSTAR = ZCAP
      VZSTAR = VZCAP
      CZSTAR = CZCAP
      GO TO 40
  170 IF ( IBOUND .EQ. 1 ) GO TO 180
      J = JSTAR
      GO TO 210
  180 VSTAR = VSTAR + P(S)
      Y(S) = ISTAR
  190 IF ( Y(JSTAR) .EQ. MP1 ) GO TO 200
      JSTAR = JSTAR + 1
      GO TO 190
  200 MB = 0
      KB = 0
      IBOUND = 0
      I = ISTAR
      Z(I) = ZSTAR
      VZ(I) = VZSTAR
      CZ(I) = CZSTAR
      GO TO 50
  210 IF ( J .GT. N ) RETURN
      MB = 0
      DO 220 I=1,M
        IFB(I) = 0
        IF ( A(I,J) .EQ. 0 ) GO TO 220
        MB = MB + 1
        B(MB) = I
        IFB(I) = 1
  220 CONTINUE
      KB = 0
```

```
      IF ( MB .LE. 1 ) GO TO 240
      IBOUND = 1
      S = J
      D = - INF
      JSTAR = J + 1
      DO 230 I=1,M
        IF ( Z(I) .GE. JSTAR ) GO TO 230
        Z(I) = JSTAR
        IF ( IFB(I) .EQ. 0 ) GO TO 230
        VZ(I) = VZ(I) + P(S)
        CZ(I) = CZ(I) - W(S)
  230 CONTINUE
      GO TO 40
  240 IF ( MB .EQ. 0 ) GO TO 250
      I = B(1)
      VSTAR = VSTAR + P(J)
      Y(J) = I
      IF ( J .LT. Z(I) ) GO TO 250
      Z(I) = J + 1
      CZ(I) = CZ(I) - W(J)
      VZ(I) = VZ(I) + P(J)
  250 J = J + 1
      GO TO 210
      END

      SUBROUTINE A(N,P,W,M,C,VSTAR,Y,CR)
      INTEGER P(N),W(N),C(M),Y(N),CR(M),VSTAR
      INF = 999999999
      VSTAR = 0
      I = 1
      J = N
      IBAR = 1
      DO 10 KK=1,M
        CR(KK) = C(KK)
   10 CONTINUE
      MP1 = M + 1
      P(N+1) = 0
      W(N+1) = INF
   20 IF ( Y(J) .EQ. MP1 ) GO TO 40
      IF ( W(J) .GT. CR(I) ) GO TO 30
      Y(J) = I
      CR(I) = CR(I) - W(J)
      VSTAR = VSTAR + P(J)
      GO TO 40
   30 I = I + 1
      IF ( I .GT. M ) I = 1
      IF ( I .NE. IBAR ) GO TO 20
      Y(J) = MP1
      I = I - 1
   40 J = J - 1
      IF ( J .EQ. 0 ) GO TO 50
      I = I + 1
      IF ( I .GT. M ) I = 1
      IBAR = I
      GO TO 20
   50 MAXC = CR(1)
      IMAXC = 1
      DO 60 I=2,M
        IF ( CR(I) .LE. MAXC ) GO TO 60
        MAXC = CR(I)
        IMAXC = I
   60 CONTINUE
      DO 80 J=1,N
        IF ( Y(J) .LT. MP1 ) GO TO 80
        IF ( W(J) .GT. MAXC ) GO TO 80
        CR(IMAXC) = CR(IMAXC) - W(J)
        VSTAR = VSTAR + P(J)
        Y(J) = IMAXC
        MAXC = CR(1)
        IMAXC = 1
        DO 70 I=2,M
          IF ( CR(I) .LE. MAXC ) GO TO 70
          MAXC = CR(I)
          IMAXC = I
   70   CONTINUE
   80 CONTINUE
      RETURN
      END

      SUBROUTINE I1(N,P,W,M,VSTAR,Y,CR)
      INTEGER P(N),W(N),Y(N),CR(M),VSTAR
C     INTEGER F(M+1,M+1)
      INTEGER F(101,101),CP,WP,FF,U,T,Q,R,S,D
```

```
      INF = 999999999
      MP1 = M + 1
      CR(MP1) = 0
      MAXF = 0
      CP = 0
      DO 20 I=1,M
        IP1 = I + 1
        DO 10 J=IP1,MP1
          F(I,J) = CR(I) + CR(J)
          F(J,I) = F(I,J)
          IF ( F(I,J) .LE. MAXF ) GO TO 10
          MAXF = F(I,J)
          IP = I
          JP = J
   10   CONTINUE
        F(I,I) = 0
        IF ( CP .LT. CR(I) ) CP = CR(I)
   20 CONTINUE
      F(MP1,MP1) = 0
      DO 30 J=1,N
        IF ( Y(J) .LT. MP1 ) GO TO 30
        FF = J
        GO TO 40
   30 CONTINUE
      RETURN
   40 WP = W(FF)
      IF ( FF .EQ. N ) GO TO 60
      IF1 = FF + 1
      DO 50 J=IF1,N
        IF ( Y(J) .LT. MP1 ) GO TO 50
        IF ( W(J) .LT. WP ) WP = W(J)
   50 CONTINUE
   60 IF ( F(IP,JP) .LT. WP ) RETURN
      J = 1
   70 IF ( CR(Y(J)) + CP .LT. WP ) GO TO 230
      K = J + 1
   80 IF ( K .GT. N ) GO TO 230
      IF ( F(Y(J),Y(K)) .LT. WP ) GO TO 120
      IF ( W(J) - W(K) ) 90,120,100
   90 U = K
      T = J
      GO TO 110
  100 U = J
      T = K
  110 D = W(U) - W(T)
      I = Y(U)
      IYT = Y(T)
      IF ( D .GT. CR(IYT) )GO TO 120
      IF ( CR(I) + D .GE. WP ) GO TO 130
  120 K = K + 1
      GO TO 80
  130 ICIPD = CR(I) + D
      MAXP = 0
      DO 140 Q=FF,N
        IF ( Y(Q) .LT. MP1 ) GO TO 140
        IF ( W(Q) .GT. ICIPD ) GO TO 140
        IF ( P(Q) .LE. MAXP ) GO TO 140
        R = Q
        MAXP = P(R)
  140 CONTINUE
      CR(I) = CR(I) + D - W(R)
      CR(IYT) = CR(IYT) - D
      VSTAR = VSTAR + P(R)
      DO 150 Q=1,M
        F(I,Q) = CR(I) + CR(Q)
        F(Q,I) = F(I,Q)
        F(IYT,Q) = CR(IYT) + CR(Q)
        F(Q,IYT) = F(IYT,Q)
  150 CONTINUE
      F(I,I) = 0
      F(IYT,IYT) = 0
      IF ( I .EQ. IP ) GO TO 160
      IF ( I .EQ. JP ) GO TO 160
      IF ( IYT .EQ. IP ) GO TO 160
      IF ( IYT .NE. JP ) GO TO 190
  160 MAXF = 0
      DO 180 Q=1,M
      IP1 = Q + 1
      DO 170 S=IP1,MP1
        IF ( F(Q,S) .LE. MAXF ) GO TO 170
        MAXF = F(Q,S)
        IP = Q
        JP = S
```

```
  170   CONTINUE
  180 CONTINUE
  190 Y(R) = I
      Y(U) = IYT
      Y(T) = I
      IF ( W(R) .NE. WP ) GO TO 210
      WP = INF
      DO 200 S=FF,N
        IF ( Y(S) .LT. MP1 ) GO TO 200
        IF ( W(S) .LT. WP ) WP = W(S)
  200 CONTINUE
  210 IF ( F(IP,JP) .LT. WP ) RETURN
      CP = 0
      DO 220 S=1,M
        IF ( CP .LT. CR(S) ) CP = CR(S)
  220 CONTINUE
      IF ( CR(Y(J)) + CP .LT. WP ) GO TO 230
      K = K + 1
      GO TO 80
  230 IF ( J .EQ. N ) RETURN
      J = J + 1
      GO TO 70
      END

      SUBROUTINE I2(N,P,W,M,VSTAR,Y,CR)
      INTEGER P(N),W(N),Y(N),CR(M),VSTAR
C     INTEGER MIN(N    ),X(N    )
      INTEGER MIN(1000),X(1000),F,T,V,CB,U,S
      INF = 999999999
      MP1 = M + 1
      MINK = INF
      MIN(N) = MINK
      DO 10 I=2,N
        KK = N + 2 - I
        IF ( W(KK) .LT. MINK ) MINK = W(KK)
        MIN(KK-1) = MINK
   10 CONTINUE
      DO 20 J=1,N
        IF ( Y(J) .LE. M ) GO TO 20
        F = J
        GO TO 30
   20 CONTINUE
      RETURN
   30 S = N
      J = N
   40 IF ( Y(J) .EQ. MP1 ) GO TO 140
      CB = CR(Y(J)) + W(J)
      IF ( CB*P(F)/W(F) .LE. P(J) ) GO TO 140
      IF ( CB .GE. W(F) ) GO TO 50
      IF ( CB .LT. MIN(F) ) GO TO 140
   50 K = F
      T = 0
      V = 0
   60 IF ( W(K) .GT. CB ) GO TO 70
      V = V + P(K)
      CB = CB - W(K)
      T = T + 1
      X(T) = K
      IF ( CB .LT. MIN(K) ) GO TO 100
   70 IF ( K .EQ. N ) GO TO 100
      K1 = K + 1
      DO 80 U=K1,N
        IF ( Y(U) .LE. M ) GO TO 80
        K = U
        GO TO 90
   80 CONTINUE
      GO TO 100
   90 IF ( V + CB*P(K)/W(K) .GT. P(J) ) GO TO 60
  100 IF ( V .LE. P(J) ) GO TO 140
      S = J
      CR(Y(J)) = CB
      DO 110  K=1,T
        Y(X(K)) = Y(J)
  110 CONTINUE
      Y(J) = MP1
      VSTAR = VSTAR + V - P(J)
      IF ( J .GT. F ) GO TO 120
      F = J
      GO TO 140
  120 IF ( Y(F) .EQ. MP1 ) GO TO 140
      IF1 = F + 1
      DO 130 U=IF1,N
        IF ( Y(U) .LE. M ) GO TO 130
```

```
      F = U
      GO TO 140
130 CONTINUE
140 J = J - 1
    IF ( J .EQ. 0 ) J = N
    IF ( J .EQ. S ) RETURN
    GO TO  40
    END
```

References

[1] Fisk, J. C., Hung, M. S.: A heuristic routine for solving large loading problems. Presented at the TIMS/ORSA Joint National Meeting, New Orleans, May 1979.

[2] Hung, M. S., Fisk, J. C.: An algorithm for 0–1 multiple knapsack problems. Naval Research Logistics Quarterly *25*, 571—579 (1978).

[3] Martello, S., Toth, P.: Algorithm for the solution of the 0–1 single knapsack problem. Computing *21*, 81—86 (1978).

[4] Martello, S., Toth, P.: Solution of the 0–1 multiple knapsack problem. Europ. J. Operat. Res. *4*, 276—283 (1980).

[5] Martello, S., Toth, P.: A bound and bound algorithm for the zero-one multiple knapsack problem. Discrete Applied Mathematics (to appear).

S. Martello
P. Toth
Istituto di Automatica
Università degli Studi di Bologna
Viale Risorgimento 2
I-40136 Bologna
Italy