ELSEVIER

European Journal of Operational Research 106 (1998) 177–183

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

Theory and Methodology

Some exact algorithms for the knapsack sharing problem ☆

Takeo Yamada *, Mayumi Futakawa [1], Seiji Kataoka

Department of Computer Science, The National Defense Academy, Yokosuka, Kanagawa 239, Japan

Received 13 June 1996; accepted 8 April 1997

Abstract

The knapsack sharing problem (KSP) is formulated as an extension to the ordinary knapsack problem. The KSP is $\mathcal{NP}$-hard. We present a branch-and-bound algorithm and a binary search algorithm to solve this problem to optimality. These algorithms are implemented and computational experiments are carried out to analyse the behavior of the developed algorithms. As a result, we find that the binary search algorithm solves KSPs with up to 20000 variables in less than a minute in our computing environment. © 1998 Elsevier Science B.V.

Keywords: Knapsack sharing problem; Combinatorial optimization; Binary search method; Max-min objective function

1. Introduction

The *knapsack problem* (KP) is fundamental in combinatorial optimization and the literature abounds on this and related subjects (see [12] for a comprehensive list of references). In the current paper we discuss the following variation of the problem. We are given a set of *items* $N := \{1, 2, \ldots, n\}$ with *weight* w_j and *value* p_j associated with each item $j \in N$. Every item belongs to one and only one of r disjoint *classes* $N_1, N_2, \ldots, N_r$, where

$$\bigcup_{k=1}^{r} N_k = N, \qquad N_k \cap N_l = \emptyset, \quad k \neq l. \tag{1}$$

Also given is a knapsack of *capacity* $\bar{c}$. As in the ordinary (0–1) KP, we wish to determine a set of items to be included in the knapsack. Thus, we introduce a vector $x = (x_j)$ of the following meaning: $x_j = 1$ if item j is selected, and $x_j = 0$ otherwise.

Our purpose is to maximize the *minimum* of $\{p^k(x) \mid k = 1, \ldots, r\}$ under the ordinary capacity constraint, where

$$p^k(x) := \sum_{j \in N_k} p_j x_j \tag{2}$$

represents the total value of the k-th class items in the knapsack. Therefore, mathematically the *knapsack sharing problem* (KSP) is formulated as follows.

(KSP)

$$\text{Maximize} \quad \min_{1 \leqslant k \leqslant r} \left\{ \sum_{j \in N_k} p_j x_j \right\}$$

$$\text{subject to} \quad \sum_{j \in N} w_j x_j \leqslant \bar{c},$$

$$x_j \in \{0, 1\}, \quad j \in N.$$

☆ Paper presented at IFORS'96, Vancouver, Canada.

* Corresponding author. Fax: +81 468 44 5911, e-mail: yamada@cs.nda.ac.jp

[1] Currently with Communication Security Group, Maritime Self-Defense Force, Japan.

0377-2217/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.

PII S0377-2217(97)00165-3

Here, without much loss of generality we assume the following.

A_0: $\bar{c}$ and p_j, w_j $(j \in N)$ are all positive integers.

KSP is $\mathcal{NP}$-hard, since for $r = 1$ the problem reduces to KP which is already $\mathcal{NP}$-hard [12].

Such a problem arises in the following situation. Suppose a city authority is going to distribute the budget of $\bar{c}$ over n possible projects involving r districts of the city. Each project is related to one and only one district, and w_j and p_j represent, respectively, the cost and benefit of the j-th project. Here, N_k is the set of projects related to the k-th district, and (2) gives the total benefit to this district. The city wishes to decide the set of projects to be accepted so as to satisfy all districts as much as possible. A reasonable solution in such a situation is to maximize the minimum benefit of all the districts; hence we have KSP.

To clarify the relation of this problem to earlier works we introduce the notation KSP($\alpha n/r/m$), where α is either one of C (meaning *continuous*), I (*integer*) or B (*binary*). This means that there are n variables of type α divided in r classes, with m constraints. Our problem is KSP($Bn/r/1$) in this notation. The special case of $r = n$ is well studied: Jacobsen [7] formulated KSP($In/n/1$) with a nonlinear objective function. Porteus and Yormark [14] gave a *binary search* algorithm to solve the same problem. Such a problem (with a linear or nonlinear objective function) is also referred to as the *max-min* or *bottleneck* allocation problem. See [8], [16], [15], [10] and [11] for this type of problem. Brown [2,3] first called these 'knapsack sharing' problems. We use the same terminology in a broader sense of $r \neq n$. Except for Kuno et al. [9] which dealt with KSP($Cn/r/1$), this case appears to have been rarely explored.

KSP can be solved by *dynamic programming* in the following way. Suppose that $x_1, x_2, \ldots, x_{j-1}$ have been (somehow) determined and we are now in the position to decide x_j $(1 \leqslant j \leqslant n)$. Let c be the remaining knapsack capacity and p^k be the total value of the class k items accumulated in the knapsack. We say that we are in *state* $(c, p^1, \ldots, p^r)$ at *stage* j. Then, denoting by $z_j^*(c, p^1, \ldots, p^r)$ the optimal objective value to go from this state and applying the *principle of optimality* [1] we obtain a recurrence relation with respect to $z_j^*(\cdot, \cdot, \ldots, \cdot)$. This can be solved in $O(n\bar{c}|\bar{p}|^r)$ steps, where

$$\bar{p} := \min_{1 \leqslant k \leqslant r} \bar{p}^k, \tag{3}$$

with

$$\bar{p}^k := \sum_{j \in N_k} p_j, \quad j = 1, 2, \ldots, r. \tag{4}$$

This approach requires $O(n\bar{c}|\bar{p}|^r)$ words of computer memory, which amounts to 100×10^6 words even for a small instance with $n = 20$, $r = 2$, $\bar{c} = 500$, $\bar{p} = 100$. Thus, by the dynamic programming algorithm we can solve only very small instances.

In the current paper, we present an upper bound and a lower bound to KSP, and based on these develop a branch-and-bound algorithm and a binary search algorithm to solve the problem to optimality. The algorithms are implemented and computational experiments are carried out to analyse the behavior of the developed algorithms. As a result, we find that the binary search algorithm solves KSPs with up to 20000 variables in less than a minute in our computing environment.

2. Branch-and-bound method

The standard branch-and-bound procedure can be tailored to solve KSP.

2.1. Subproblems

First, corresponding to an arbitrary partition $Q = (Q_0, Q_1, Q_X)$ of N we introduce a subproblem of KSP by fixing the variables in Q_0 (Q_1, resp.) at 0 (1). Thus we have

(Subproblem Q)

$$\begin{aligned} \text{Maximize} \quad & \min_{1 \leqslant k \leqslant r} \left\{ \sum_{j \in N_k} p_j x_j \right\} \\ \text{subject to} \quad & \sum_{j \in N} w_j x_j \leqslant \bar{c}, \\ & x_j \in \{0, 1\}, \quad j \in Q_X, \\ & x_j = 0, \quad j \in Q_0, \\ & x_j = 1, \quad j \in Q_1. \end{aligned}$$

Q is said to be a *terminal* subproblem if $Q_X = \emptyset$. Or, Q is *infeasible* if $\sum_{j\in Q_1} p_j > \bar{c}$. If either one of these happens, Q is terminated. Subproblem Q is also terminated if it has been proved unpromising by upper bounds. We discuss such an upper bound in the following subsection. If none of these occurs, for an arbitrary item $j \in Q_X$ we can construct two *children* of Q by removing j from Q_X and adding it to either one of Q_0 or Q_1.

An appropriate choice of such an item j, or the *branching strategy*, greatly affects the efficiency of the resulting branch-and-bound algorithm. Here we mention the following two strategies.

1. *Sequential branching:* variables are picked up sequentially from x_1, x_2 through x_n.

2. *Min-first branching:* from the class of the least total accumulated value in the knapsack we pick up the first unfixed variable.

After some preliminary numerical tests, we found the sequential strategy superior to the min-first both in CPU time and the number of subproblems generated in the process of computation.

2.2. An upper bound

Subproblem Q can be transformed into the standard KSP form as follows. First, we delete from the subproblem all fixed items in $Q_0 \cup Q_1$ and add to each class k an item of weight 0 and value $\sum_{j\in N_k\cap Q_1} p_j$, provided $N_k \cap Q_1 \neq \emptyset$. The knapsack capacity is modified to $\bar{c} - \sum_{j\in Q_1} w_j$. Note that subproblems in this form may include some items of zero weight. Therefore, in what follows we replace assumption A_0 with the following.

A_1: $\bar{c}$ and p_j $(j \in N)$ are positive integers, and w_j $(j \in N)$ are non-negative integers.

Then, to obtain upper and lower bounds for subproblems it suffices to discuss those for KSP. Before doing so, we define the *efficiency* of an item as the value of that item per unit weight. Thus for item j this is $e_j := p_j/w_j$. Without loss of generality, we further introduce the following assumptions.

A_2: There exist integers $0 = t_0 < t_1 < \cdots < t_r = n$ such that $N_k \equiv \{t_{k-1}+1, t_{k-1}+2, \ldots, t_k\}$, $k = 1, 2, \ldots, r$.

A_3: Within each class, items are numbered in non-increasing order of efficiency; i.e. $i, j \in N_k$, $i < j$ implies $e_i \geqslant e_j$.

For simplicity we adopt the following abbreviation:

$$w_{k,j} := w_{t_{k-1}+j}, \qquad p_{k,j} := p_{t_{k-1}+j}. \tag{5}$$

Now, to obtain an upper bound to KSP we solve the following *continuous relaxation* of KSP.

(C(KSP))

$$\begin{aligned} \text{Maximize} \quad & \min_{1\leqslant k\leqslant r} \Big\{ \sum_{j\in N_k} p_j x_j \Big\} \\ \text{subject to} \quad & \sum_{j\in N} w_j x_j \leqslant \bar{c}, \\ & 0 \leqslant x_j \leqslant 1, \quad j \in N. \end{aligned}$$

We introduce an auxiliary problem to solve C(KSP) through decomposition.

($\mathrm{Aux}_k(c^k)$)

$$\begin{aligned} \text{Maximize} \quad & \sum_{j\in N_k} p_j x_j \\ \text{subject to} \quad & \sum_{j\in N_k} w_j x_j \leqslant c^k, \\ & 0 \leqslant x_j \leqslant 1, \quad j \in N_k. \end{aligned}$$

Note that this is the continuous KP, which is easy to solve [12]. Let $\bar{z}^k(c^k)$ denote the optimal objective value to $\mathrm{Aux}_k(c^k)$, and define

$$w_j^k := \sum_{i=1}^{j} w_{k,i}, \qquad p_j^k := \sum_{i=1}^{j} p_{k,i}, \tag{6}$$

for $j = 1, \ldots, |N_k|$, as the *cumulative* weight (profit) of the k-th class items, where by (4) we have $p^k_{|N_k|} \equiv \bar{p}^k$. Then, $\bar{z}^k(\cdot)$ is given as the piece-wise linear function obtained by connecting the points $\{(w_j^k, p_j^k) \mid j = 0, 1, \ldots, |N_k|\}$, as depicted in Fig. 1. From assumptions A_1 and A_3, $\bar{z}^k(\cdot)$ is concave and monotonically increasing with $\bar{z}^k(0) = 0$. The inverse of this function, denoted as $\bar{c}^k(\cdot)$, is defined over $[0, \bar{p}^k]$. We note $\bar{p} \leqslant \bar{p}^k$, for $k = 1, 2, \ldots, r$.

In solving C(KSP) we first assume $\sum_{k=1}^{r} \bar{c}^k(\bar{p}) \leqslant \bar{c}$. Then, solving $\mathrm{Aux}_k(c^k)$ with $c^k := \bar{c}^k(\bar{p})$ for $k = 1, 2, \ldots, r$, we immediately obtain an optimal solution to C(KSP) with the corresponding objective value $\bar{p}$.

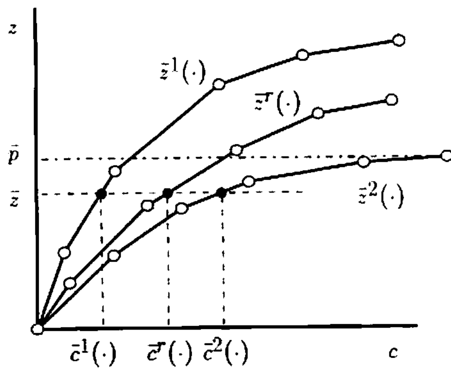

Fig. 1. Graphical representation of $\bar{\mathrm{C}}$(KSP).

Next, we consider the following case:

A_4: $\sum_{k=1}^{r} \bar{c}^k(\bar{p}) > \bar{c}$.

We note that C(KSP) is equivalent to the following.

($\bar{\mathrm{C}}$(KSP))

$$\begin{aligned} &\text{Maximize} \quad \min_{1\leqslant k\leqslant r} \{\bar{z}^k(c^k)\} \\ &\text{subject to} \quad \sum_{1\leqslant k\leqslant r} c^k \leqslant \bar{c}, \\ &\qquad\qquad c^1,\ldots,c^k \geqslant 0. \end{aligned}$$

Under the assumptions A_1–A_4, the following gives the first order necessary condition for $(\bar{c}^1, \bar{c}^2, \ldots, \bar{c}^r)$ to be an optimal solution to $\bar{\mathrm{C}}$(KSP) with the corresponding objective value $\bar{z}$ (see Fig. 1):

$$\begin{aligned} &\bar{z}^k(\bar{c}^k) = \bar{z}, \quad k = 1,\ldots,r, \\ &\sum_{k=1}^{r} \bar{c}^k = \bar{c}. \end{aligned} \tag{7}$$

Clearly, $\bar{z}$ gives an upper bound to KSP. Kuno et al. [9] gave a linear time algorithm to find $(\bar{c}^1, \bar{c}^2, \ldots, \bar{c}^r)$ as well as $\bar{z}$. With these, the solution to $\mathrm{Aux}_k(\bar{c}^k)$ is given by

$$\bar{x}_{k,j} := \begin{cases} 1 & \text{if } j < u_k, \\ (\bar{c}^k - w_{j-1}^k)/w_{k,j} & \text{if } j = u_k, \\ 0 & \text{if } j > u_k, \end{cases} \tag{8}$$

where

$$u_k := \min\{j \mid p_j^k \geqslant \bar{z}\}. \tag{9}$$

2.3. A lower bound

Using the above obtained $u_1, u_2, \ldots, u_r$, we define a feasible solution to KSP by

$$\underline{x}_{k,j} := \begin{cases} 1 & \text{if } j < u_k, \\ 0, & \text{otherwise}, \end{cases} \quad k = 1,\ldots,r. \tag{10}$$

This is referred to as the *trivial solution* to KSP. Starting with this we can further improve the solution by successively filling that part of the knapsack which is left vacant in the following way: at each step we compare the total value of the items of each class in the knapsack, and from the class of smallest value we adopt the feasible item of highest efficiency into the knapsack. We repeat this until no further improvement is possible this way. The solution resulting from this algorithm is referred to as the *greedy solution*, and the corresponding objective value is denoted as $\underline{z}$. This gives a lower bound to KSP. The computational complexity of this part is $O(n)$.

3. Binary search method

Let z^* be the optimal objective value to KSP. For an arbitrary integer $z \geqslant 0$, we have $z^* \geqslant z$ if and only if

$$\begin{aligned} &\sum_{j\in N_k} p_j x_j \geqslant z, \quad k = 1,\ldots,r, \\ &\sum_{j\in N} w_j x_j \leqslant \bar{c}, \\ &x_j \in \{0,1\}, \quad j \in N, \end{aligned} \tag{11}$$

is feasible. Thus, we say that z is *attainable* if (11) is feasible, and *unattainable* otherwise. Let us introduce the *inverse knapsack problem* (IKP) as follows.

($\mathrm{IKP}_k(z)$)

$$\begin{aligned} &\text{Minimize} \quad \sum_{j\in N_k} w_j x_j \\ &\text{subject to} \quad \sum_{j\in N_k} p_j x_j \geqslant z, \\ &\qquad\qquad x_j \in \{0,1\}, \quad j \in N_k. \end{aligned}$$

With $c^{*k}(z)$ denoting the optimal objective value to $\mathrm{IKP}_k(z)$, attainability of z is equivalent to

$$\sum_{k=1}^{r} c^{*k}(z) \leqslant \bar{c}. \tag{12}$$

Note that by a simple change of variables from x_j to $y_j := 1 - x_j$ ($j \in N$), $\mathrm{IKP}_k(z)$ is transformed into a standard 0–1 KP. Let $\underline{c}^k(z)$ and $\bar{c}^k(z)$ be, respectively, a lower and an upper bounds to $\mathrm{IKP}_k(z)$. These can be obtained as the upper and lower bounds of the previous section with $r = 1$. Alternatively, several methods of evaluating these bounds can be found in some textbooks on combinatorial optimization (see, e.g., [13]). Then from (12), without calculating the exact $c^{*k}(z)$ we can judge z attainable if

$$\sum_{k=1}^{r} \bar{c}^k(z) \leqslant \bar{c}, \tag{13}$$

and unattainable if

$$\sum_{k=1}^{r} \underline{c}^k(z) > \bar{c}. \tag{14}$$

If neither of these holds, we need to have $c^{*k}(z)$ which is certainly time consuming. However, since algorithms for the 0–1 KP are well explored and efficient codes are available in the literature (such as Horowitz–Sahni [6] or Martello–Toth [12]), this can be obtained in reasonable CPU time for considerably large-sized problems.

In the following binary search algorithm, $\{z_L, z_R\}$ is a pair of attainable and unattainable values, which is initially set to $\{\underline{z}, \bar{z}\}$ as defined in the previous section. Here we note that $\underline{z}$ is attainable and $\bar{z}$ can be assumed unattainable, since otherwise the problem is obviously solved by $z^* := \bar{z}$. At each iteration, the *midpoint* is checked for attainability and the interval is halved accordingly. The algorithm is terminated as soon as $z_R - z_L = 1$ is reached, where $z^* := z_L$ gives the optimal objective value. Thus we have:

procedure *Binary-Search*;
  input $\underline{z}, \bar{z}$;
  output z^*;
begin
  $z_L := \underline{z}$; $z_R := \bar{z}$; {Initialization}
  repeat
    $z := \lfloor (z_L + z_R)/2 \rfloor$;
    if $\sum_{k=1}^{r} \underline{c}^k(z) > \bar{c}$ **then** $z_R := z$;
    else if $\sum_{k=1}^{r} \bar{c}^k(z) \leqslant \bar{c}$ **then** $z_L := z$;
    else
      begin
        if $\sum_{k=1}^{r} c^{*k}(z) > \bar{c}$ **then** $z_R := z$;
        else $z_L := z$;
      end;
  until $z_R - z_L = 1$;
  $z^* := z_L$;
end.

4. Numerical experiments

We have implemented the branch-and-bound and binary search algorithms of the previous sections in C language and conducted a series of computational experiments on a DECsystem 300/400 workstation. This section describes the results of the experiments.

4.1. Test problems

First, we consider the case of two classes with $n_1 = n_2 = \frac{1}{2}n$ items each. Weights and values of items are randomly distributed according to:

- *Uncorrelated case:*
  w_j: uniformly random in [1, 1000],
  p_j: uniformly random in [1, 1000].
- *Weakly correlated case:*
  w_j: uniformly random in [1, 1000],
  p_j: uniformly random in $[w_j, w_j + 200]$.
- *Strongly correlated case:*
  w_j: uniformly random in [1, 1000],
  $p_j := w_j + 100$.

Cases of more than two classes will be briefly mentioned in the following subsection.

4.2. Results of experiments

Tables 1(a)–(c) summarize the results of computation, where CPU times are compared between the branch-and-bound (B&B) and binary search (Binary) algorithms. Corresponding to n from 200 up to 80 000, each row gives the average of 10 randomly generated test problems with $\bar{c} = 250n$. Tables 1(a), 1(b) and 1(c) correspond, respectively, to the uncorrelated, weakly correlated and strongly correlated cases. In these tables, #subprobs gives the number of subproblems produced in the branch-and-bound process, and #IKP stands for the number of IKPs solved to optimality using the Horowitz–Sahni algorithm [6].

Table 1
Computational results for three cases

n	B&B		Binary	
	# subprobs	CPU sec	#IKP	CPU sec
(a) Uncorrelated case				
200	8 886.0	2.15	12.10	0.03
400	12 951.8	4.58	11.30	0.06
600	14 288.8	7.42	11.10	0.13
800	23 529.6	15.29	10.80	0.20
1 000	41 255.6	35.20	9.30	0.20
1 500	107 241.0	122.95	10.70	0.57
2 000	82 260.8	122.67	9.50	0.79
5 000	141 458.3[8]	546.48[8]	9.30	4.72
10 000	142 733.0[5]	1051.50[5]	7.30	12.33
20 000	-	-	7.70	48.30
50 000	-	-	5.20	203.78
80 000	-	-	3.90	407.34
(b) Weakly correlated case				
200	55 090.4	13.64	11.80	0.06
400	452 119.0	244.81	11.10	0.16
600	298 136.8	217.68	9.50	0.25
800	229 646.2	210.11	10.10	0.40
1 000	149 327.6	174.47	7.70	0.32
1 500	368 943.0	630.83	8.30	0.58
2 000	322 010.5[8]	708.84[8]	8.90	0.94
5 000	182 465.3[7]	809.10[7]	6.40	3.09
10 000	156 665.5[4]	1371.78[4]	6.00	9.84
20 000	-	-	4.30	26.42
50 000	-	-	2.60	101.94
80 000	-	-	1.20	125.48
(c) Strongly correlated case				
200	2 005 653.0[4]	411.47[4]	8.70	41.60
400	-	-	10.67[3]	550.63[3]

Superscribed are the number of problems solved (out of 10) within the CPU time limit of 1800 seconds, while this is omitted if all 10 problems are solved. Dashes (-) indicate that no instances were solved within the time limit. Averages are taken only for the problems solved within this time limit.

From these tables we observe that the binary search method outperforms the branch-and-bound algorithm for all classes of test problems. For the uncorrelated and weakly correlated cases, the binary search algorithm solves problems of up to 20000 items in less than a minute, while with the branch-and-bound algorithm this can be pretty hard even for the problems of a few hundred items. The strongly correlated case is difficult to solve with either of the algorithms. One remarkable observation with the binary search algorithm is the fact that with increasing size of instances, we usually need to solve less IKPs to optimality.

Table 2
Cases of more than two classes of items

r	n	Uncorrelated		Weakly correlated	
		#IKP	CPU sec	#IKP	CPU sec
3	200	18.00	0.22	18.20	0.72
	400	16.00	0.45	16.50	1.35
	600	16.20	0.69	15.50	2.52
	800	14.30	0.80	14.10	2.99
	1 000	16.50	1.39	14.10	3.78
	2 000	14.40	3.06	12.10	5.90
	5 000	12.20	7.96	11.20	11.64
	10 000	10.60	17.42	10.50	19.74
5	200	23.70	0.12	28.00	0.36
	400	31.90	0.36	27.40	1.18
	600	30.10	0.69	26.70	1.80
	800	28.30	0.90	25.70	2.56
	1 000	28.40	1.35	24.00	3.89
	2 000	24.40	2.47	21.60	6.35
	5 000	22.10	6.96	17.90	11.00
	10 000	21.50	17.31	16.60	20.05
10	200	55.20	0.09	62.20	0.33
	400	61.30	0.32	58.90	0.86
	600	62.00	0.50	51.60	1.35
	800	57.50	0.62	55.50	2.56
	1 000	57.80	0.88	52.40	3.19
	2 000	54.80	2.35	46.10	6.50
	5 000	51.10	5.83	42.30	14.48
	10 000	44.90	13.60	35.30	24.65

Next, Table 2 shows the results of the binary search algorithm applied to the case of r (> 2) classes consisting of n/r items each.

Here we observe that the more classes we have, the more IKPs we need to solve to optimality. However, CPU time remains almost the same with the increase of r. This appears to happen since (with fixed n) we have smaller-sized IKPs for larger r.

5. Conclusion

We have formulated the knapsack sharing problem as an extension to the knapsack problem, and presented two algorithms to solve the problem to optimality. The algorithms were implemented and compu-

tational experiments were carried out. As a result of the experiments, we found that the binary search algorithm solves KSP with 20000 variables in less than a minute in our computing environment.

The following items are left for future research. Performance of a branch-and-bound procedure depends highly on particular implementation of that algorithm. Programs of this kind are often accelerated drastically by employing sophisticated implementation techniques. The computer code developed in this research is rather a crude realization of the branch-and-bound method; thus, refinement of the algorithm in this direction should certainly be pursued. The binary search method may further be accelerated by adopting more advanced methods (such as MT2 or BZ in [12], which is claimed to be faster than the Horowitz–Sahni algorithm [6] used in the current work) to solve IKP more rapidly. Finally, more experiments for larger instances should be conducted. For such a problem, (meta-)heuristic algorithms such as simulated annealing or tabu search might prove useful.

References

[1] Bellman, R., Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.

[2] Brown, J.R., The knapsack sharing problem, Operations Research 27 (1979) 341–355.

[3] Brown, J.R., Bounded knapsack sharing, Mathematical Programming 67 (1994) 343–382.

[4] Chvátal, V., Linear Programming, Freeman, New York, 1983.

[5] Garey, M.R., Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

[6] Horowitz, E., Sahni, S., Computing partitions with applications to the knapsack problem, Journal of the ACM 21 (1974) 277–292.

[7] Jacobsen, S., On marginal allocation in single constraint min-max problems, Management Science 17 (1971) 780–783.

[8] Kaplan, S., Application of programs with maximin objective functions to problems of optimal resource allocation, Operations Research 22 (1974) 802–807.

[9] Kuno, T., Konno, H., Zemel, E., A linear-time algorithm for solving continuous maximin knapsack problems, Operations Research Letters 10 (1991) 23–26.

[10] Luss, H., A nonlinear minimax allocation problem with multiple knapsack constraints, Operations Research Letters 10 (1991) 183–187.

[11] Luss, H., Minimax resource allocation problems: optimization and parametric analysis, European Journal of Operational Research 60 (1992) 76–86.

[12] Martello, S., Toth, P., Knapsack Problems: Algorithms and Computer Implementations, Wiley, New York, 1990.

[13] Nemhauser, G.L., Wolsey, L.A., Integer and Combinatorial Optimization, Wiley, New York, 1988.

[14] Porteus, E.L., Yormark, J.S., More on min-max allocation, Management Science 18 (1972) 502–507.

[15] Tang, C.S., A max-min allocation problem: its solutions and applications, Operations Research 36 (1988) 359–367.

[16] Zeitlin, Z., Integer allocation problems of min-max type with quasiconvex separable functions, Operations Research 29 (1981) 207–211.