ISSN 0918-7553

Research Report

On solving the max-min 0–1 knapsack problem

Hiroshi Iida

June 2, 1997
IS-RR-97-0025F

JAIST

School of Information Science

Japan Advanced Institute of Science and Technology, Hokuriku

On solving the max-min 0–1 knapsack problem

Hiroshi Iida
June 2, 1997
IS-RR-97-0025F

School of Information Science
JAIST, Hokuriku
1–1 Asahidai, Tatsunokuchi,
Ishikawa 923–12, Japan
x-iida@jaist.ac.jp

ISSN 0918-7553

Abstract

The max-min 0–1 knapsack problem was recently introduced by Gang Yu in the journal of *Operations Research*. The problem is an extension of the classical 0–1 knapsack problem. This report includes several suggestions on the paper in which upper and lower bounds for the max-min 0–1 knapsack problem have been proposed. In this report we also propose new upper and lower bounds and a new algorithm which exploits our proposed bounds. Computation experiments are also included.

Keywords: knapsack problem; greedy heuristic; surrogate relaxation; Lagrangian relaxation; branch-and-bound method

1 Introduction

The max-min 0–1 knapsack problem has been introduced by Yu [5], which is formulated as follows:

$$z_{\mathrm{MNK}} = \max_{x} \min_{s \in S} \left\{ \sum_{i=1}^{n} v_i^s x_i \;\middle|\; \sum_{i=1}^{n} a_i x_i \leq b;\ x_i \in \{0,1\},\ i = 1, \ldots, n \right\},$$

where S is a set of scenarios and each item i has value v_i^s under scenario $s \in S$. Namely, any scenario consists of n values for each item. Also any item i has weight a_i independent of the scenarios, which satisfies $0 \leq a_i \leq b$ and $\sum_i a_i > b$. The max-min 0–1 knapsack (MNK) problem aims to select items so that minimal total value gained by the selected items under all scenarios is maximal under the constraint that total weight of the selected items does not exceed given capacity b. The MNK problem is also formulated more simply without 0–1 variables $\{x_i\}_{1 \leq i \leq n}$ as follows:

$$z_{\mathrm{MNK}} = \max_{I \subset N} \min_{s \in S} \left\{ \sum_{i \in I} v_i^s \;\middle|\; \sum_{i \in I} a_i \leq b \right\},$$

where $N := \{1, 2, \ldots, n\}$. Throughout this report, we will denote a subset of N consisting of item i fixed to $x_i = 0$ (always ignored) by N_0, and a subset of N consisting of item i fixed to $x_i = 1$ (always taken) by N_1. Also $N_2 := N \setminus (N_0 \cup N_1)$.

We often call a subset of N *solution*. In particular a solution I is said to be *feasible*, if it satisfies $\sum_{i \in I} a_i \leq b$. Sometimes we identify n-vector of 0–1 variables $x = (x_i)$ with solution $I \subseteq N$ as $x_i = 1 \Leftrightarrow i \in I$ naturally.

Example. We present a small-sized example. Consider an instance of MNK with five items. The number of scenarios is two and capacity is 126. On each item i, weight a_i and value v_i^s under scenario s are given by

i	1	2	3	4	5
a_i	3	37	84	30	99
v_i^1	16	22	12	73	106
v_i^2	39	101	4	34	94
b			126		

For instance, solution $\{1, 3, 4\}$ is feasible, because $a_1 + a_3 + a_4 = 117 \leq 126 = b$. Also the value of the solution is 77, because $v_1^1 + v_3^1 + v_4^1 = 101 > 77 = v_1^2 + v_3^2 + v_4^2$. In this example an optimal solution is $\{1, 5\}$ with optimal value 122.

In the case where the number of scenarios (We denote it by $|S|$) is one, MNK reduces to the classical 0–1 Knapsack Problem (KP). The MNK problem thus includes KP as special case and is $\mathcal{NP}$-hard. In addition when the number of scenarios is unbounded, MNK is strongly $\mathcal{NP}$-hard, which is also proved in [5]. The MNK problem has been solved by branch-and-bound with best-first search in [5], where the extreme tightness of the proposed bounds has expedited the branch-and-bound.

The remainder of this report is organized as follows: In Section 2 we discuss greedy heuristics to obtain lower bound of the MNK problem. We also discuss relaxation of MNK for upper bound in Section 3. In Section 4 we develop an algorithm to solve MNK, which exploits our proposed bounds. The last section is devoted to conclusion.

2 Greedy heuristics — for lower bound —

The lower bound proposed in [5] has been compared with greedy heuristic which is also introduced in [5] in order to demonstrate its efficiency. Although the greedy heuristic has been introduced as natural extension of the one for KP, to the best of our knowledge, it is not so. In this section we discuss four greedy heuristics which are really natural extension of the one for KP respectively.

2.1 Density-ordered

The greedy heuristic introduced in [5] takes k-th item i_k at *STEP 2* as follows:

$$i_k = \max_{i \in N_2} \frac{\min_{s \in S} \left\{ f_s^{k-1} + v_i^s \right\}}{a_i}, \tag{1}$$

where $f_s^{k-1} = v_{i_1}^s + v_{i_2}^s + \cdots + v_{i_{k-1}}^s$, which is the total value of already taken $k-1$ items under scenario s. Also weight of item i_k should be suitable to residual capacity i.e. $a_{i_k} \leq b - \sum_{j=1}^{k-1} a_{i_j}$, or a given solution is not feasible.

Then, in the case where $|S| = 1$ the formula (1) is reduced to

$$i_k = \max_{i \in N_2} \frac{f^{k-1} + v_i}{a_i}.$$

This is however corresponding to none of four greedy heuristics for Integer Knapsack Problem (IKP) discussed in Kohli & Krishnamurti [4]; weight-ordered, value-ordered, density-ordered and total-value greedy heuristics. Since IKP is an extension of KP (It is allowed to take an item more than one), any greedy heuristic for IKP is applicable to KP without modification. Therefore, at least to be the natural extension of typical density-ordered greedy heuristic for KP, the formula (1) should be replaced with

$$i_k = \max_{i \in N_2} \frac{\min_{s \in S} v_i^s}{a_i}. \tag{2}$$

In fact the greedy heuristic exploiting the formula (2) improves rather than the one exploiting the formula (1), which will be confirmed by computation experiments afterwards.

We present an example to see the difference of behavior between the greedy heuristics exploiting the formula (1) and the one exploiting (2), where $n = 5$, $|S| = 2$ and $b = 150$:

i	1	2	3	4	5
a_i	95	11	98	37	59
v_i^1	15	71	92	71	33
v_i^2	11	77	76	71	34
$\min_{s \in S} \left\{ f_s^2 + v_i^s \right\} / a_i$	1.65	—	2.28	—	2.96
$\min_{s \in S} v_i^s / a_i$	0.11	6.45	0.77	1.91	0.55
b			150		

Repeating that our aim is to find a subset $I \subset \{1,2,3,4,5\}$ which maximizes $\min_{s \in \{1,2\}} \sum_{i \in I} v_i^s$. In this example, the former (1) finds $\{2,4,5\}$ with value 175, while the latter (2) finds $\{2,3,4\}$ with optimal value 224. Each takes $\{2,4\}$ severally, however, next one is different because of

the inner estimation of each. The fifth row of the table presents the estimation by the former after taking $\{2, 4\}$, where $f_s^2 = v_2^s + v_4^s$.

Before making a comparison between the two experimentally, we will illustrate data instances provided for the comparison. It has been taken from a part of Table III in [5] ($\delta = 0.9$), where

$$\begin{array}{lcl} a_i & : & \text{uniformly distributed in } (0, 100) \\ \hat{v}_i & : & \text{base value for } v_i^s, \text{ uniformly distributed in } (0, 100) \\ v_i^s & : & \text{uniformly distributed in } (\hat{v}_i(1-\delta), \hat{v}_i(1+\delta)) \text{ for all } s \in S \\ b & : & \sum_{i=1}^{n} a_i/m. \end{array}$$

Also we provide all these data as (unsigned long) integer type in C. Throughout this report, all data instances provided for computation experiment conform to this style. It should be noted that data instance of MNK becomes hard as δ becomes large, which is mentioned in [5].

Computation experiments are in Table 1. Each line expresses the average lower bound of 100 data instances, so two heuristics are always applied to same data instances. As we can see in Table 1, the greedy heuristic exploiting the formula (2) always produces sharp lower bound rather than the one exploiting the formula (1).

Table 1: Extension of greedy heuristics for IKP

δ	n	$\lvert S\rvert$	m	formula (1)	formula (2)
0.9	60	10	2	1808.98	2009.71
			3	1469.06	1592.88
			4	1252.60	1369.28
		20	2	1758.71	1943.25
			3	1402.87	1549.25
			4	1199.79	1313.27
		30	2	1745.37	1940.16
			3	1337.38	1486.21
			4	1172.32	1293.68

2.2 Total-value and others

As mentioned previously, there exist four greedy heuristics for IKP [4]. In this subsection we will mention the extension of another one of them called *total-value* greedy heuristic and other two greedy heuristics.

Focusing on the case where $|S| = 1$, the extension of the total-value greedy heuristic for IKP should be prescribed as replacing the formula (1) with

$$i_k = \max_{i \in N_2} \min_{s \in S} v_i^s \left\lfloor \frac{b}{a_i} \right\rfloor. \tag{3}$$

Indeed when $|S| = 1$, this formula reduces to $i_k = \max_{i \in N_2} v_i \lfloor b/a_i \rfloor$, which is corresponding to the definition of total-value greedy heuristic for IKP.

We present an example to see the difference of behavior between the greedy heuristics exploiting the formula (2) and the one exploiting the formula (3):

i	1	2	3	4	5
a_i	85	91	35	90	64
v_i^1	50	69	38	42	58
v_i^2	77	14	27	36	42
$\min_{s\in S} v_i^s/a_i$	0.58	0.15	0.77	0.40	0.65
$\min_{s\in S} v_i^s\lfloor b/a_i\rfloor$	100	28	135	72	84
b			182		

In this example the former (2) finds $\{3,5\}$ with value 69, while the latter (3) finds $\{1,3\}$ with value 88. First each takes item 3 severally, however, next one is different because of the inner estimation of each, which is presented in the last two rows of the table. In passing an optimum is $\{1,5\}$ with value 108, so item 3 should not be taken in this example.

In addition we can discuss another extension of greedy heuristics for IKP, that is, taking item i according to a nonincreasing order of $\sum_{s\in S} v_i^s$. This is an extension of the *value-ordered* greedy heuristic. While this heuristic can find an optimal solution $\{1,5\}$ in the previous example, it is not promising in general (see Table 2 presented subsequently).

Here we should mention the last of the four, *weight-ordered*. The greedy heuristic will take an item according to a nondecreasing order of weight. Namely we first take item i, provided a_i is minimal. Clearly this heuristic is applicable to MNK without modification, however, it does not seem promising. For instance, on the example in the preceding subsection, it has found $\{2,4,5\}$. In addition, it has also found $\{3,5\}$ in the previous example.

Now we present a comparison among the four greedy heuristics in Table 2. Each line expresses the average lower bound of 100 data instances. As in Table 2, the extension of total-value is a little better than density-ordered. It is however an comparison of average, so sometimes the extension of density-ordered has produced better one than total-value under our observation. Since it is not so expensive to apply each heuristic to MNK respectively, we hereafter adopt the maximum between the two, density-ordered and total-value, as our enhanced greedy heuristic for the MNK problem. It is denoted by the symbol z_{EG}.

Table 2: Extension of greedy heuristics of those for IKP

δ	n	$\lvert S\rvert$	m	density-ordered*	total-value**	value-ordered	weight-ordered
0.9	60	10	2	2024.79	2026.81	1990.65	1808.06
			3	1599.40	1600.38	1496.85	1429.38
			4	1347.82	1355.45	1205.72	1192.39
		20	2	1975.65	1978.31	1919.69	1727.25
			3	1542.27	1543.28	1371.14	1374.22
			4	1296.16	1302.38	1102.83	1167.04
		30	2	1947.12	1947.86	1864.12	1701.64
			3	1517.34	1513.05	1346.45	1329.93
			4	1289.85	1292.49	1089.78	1109.66

$*$: exploiting the formula (2)
$**$: exploiting the formula (3)

3 Relaxation — for upper bound —

In this section we discuss relaxation of the MNK problem to obtain upper bound. Note that it is not a requirement for the solution which gives upper bound to be feasible on maximization

problem. Also it is extremely preferable that the gap between upper bound and optimal value is as small as possible in order to exploit branch-and-bound effectively.

Although we have exploited double precision of floating-point accumulation to obtain upper bound and round it to integer type, there exists a little opportunity that it leads to incorrect result. To cope with it, we use Unix$^{\mathrm{TM}}$ arithmetic library function ceil(3M) when rounding final result of double precision to unsigned long integer. This operation might have to be reconsidered whether there exists more accurate operation or not.

3.1 Lagrangian relaxation

In this subsection we discuss Lagrangian relaxation for MNK. While the relaxation used in [5] is based on *surrogate relaxation* proposed by Glover [3], we will show that the relaxation is indeed equivalent to Lagrangian relaxation.

As mentioned in [5], MNK can be written as the following 0–1 Integer Linear Program:

$$\begin{aligned} \text{maximize} \quad & y \\ \text{subject to} \quad & y \le \sum_{i=1}^{n} v_i^s x_i, \ \text{for all } s \in S \\ & \sum_{i=1}^{n} a_i x_i \le b, \ x_i \in \{0,1\}, \ i = 1, 2, \ldots, n. \end{aligned}$$

Using Lagrangian multipliers $\mu_s \ge 0\,(s \in S)$ with a special restriction of $\sum_{s\in S} \mu_s = 1$ also used in [5] for aggregating the $|S|$ constraints as to value, we have Lagrangian relaxation as follows (The notation of Z_D follows Fisher [2]):

$$\begin{aligned} Z_D(\mu) &= \max_x \ y + \sum_{s\in S} \mu_s \left(\sum_{i=1}^{n} v_i^s x_i - y \right) \\ &= \max_x \ \left(1 - \sum_{s\in S} \mu_s \right) y + \sum_{s\in S} \mu_s \sum_{i=1}^{n} v_i^s x_i \\ &= \max_x \ \sum_{i=1}^{n} v_i(\mu) x_i, \ \text{where } v_i(\mu) = \sum_{s\in S} \mu_s v_i^s \ (1 \le i \le n). \end{aligned}$$

The last formula is thus identical with the surrogate relaxation $z_U(\mu)$ mentioned in [5].

By this result, it is meaningless to apply Lagrangian relaxation to the $|S|$ constraints as to value in order to obtain sharp upper bound rather than the one produced by surrogate relaxation. In addition the upper bound proposed in [5] outperforms the one produced by LP (Linear Programming) relaxation in overwhelming cases. Hence we need something else in place of these three relaxations.

3.2 Extension of LP relaxation for KP

To have another upper bound, we will here consider the extension of the LP relaxation applied to KP: well-known Dantzig upper bound [1].

Under the assumptions of first two formulae in the following, the formula (4) might seem to be promising:

$$\frac{\min_{s\in S} v_1^s}{a_1} \ge \frac{\min_{s\in S} v_2^s}{a_2} \ge \cdots \ge \frac{\min_{s\in S} v_n^s}{a_n}$$

$$\sum_{i=1}^{k-1} a_i \leq b < \sum_{i=1}^{k} a_i$$
$$\min_{s\in S} \left\lfloor \sum_{i=1}^{k-1} v_i^s + \left(b - \sum_{i=1}^{k-1} a_i \right) \frac{v_k^s}{a_k} \right\rfloor . \tag{4}$$

However the formula (4) does not give upper bound. We present a counterexample of which the formula (4) does not give upper bound as follows:

i	1	2	3	4
a_i	1	1	1	1
v_i^1	7	6	5	3
v_i^2	6	5	4	5
$\min_{s\in S} v_i^s/a_i$	6	5	4	3
b	3			

Then the formula (4) finds $\{1,2,3\}$ with value 15, however, we can obtain an optimal value of $z_{\mathrm{MNK}} = 16$ which is given by $\{1,2,4\}$. Needless to say, upper bound always has to be greater than or equal to optimal value.

3.3 Relaxation mixture

In this subsection we are concerned with the surrogate (or Lagrangian) relaxation. Note that the surrogate relaxation of MNK is a mere KP. We will here denote it once again:

$$\begin{aligned} Z_D(\mu) = \max_x \ & \sum_{i=1}^{n} v_i(\mu) x_i, \ v_i(\mu) = \sum_{s\in S} \mu_s v_i^s, \text{ where } \mu_s \geq 0 \, (s \in S), \ \sum_{s\in S} \mu_s = 1 \\ \text{subject to } & \sum_{i=1}^{n} a_i x_i \leq b, \ x_i \in \{0,1\}, \ i = 1,2,\ldots,n. \end{aligned} \tag{5}$$

In [5], subgradient method is for use in obtaining bounds. In the method, the KP (5) with a fixed multiplier vector is solved exactly and the result is used to improve the multiplier vector in every iteration, however, the processing is no doubt expensive. Therefore we will apply LP relaxation technique to the KP (5), and exploit an obtained result as upper bound. We will denote the upper bound by z_{SL}.

Before appling LP relaxation to the KP (5), we should fix a multiplier vector $(\mu_s)_{s\in S}$. We will try to determine it as follows:

$$\mu_s = \frac{\dfrac{1}{\sum_{i=1}^{n} v_i^s}}{\sum_{t\in S} \dfrac{1}{\sum_{i=1}^{n} v_i^t}}. \tag{6}$$

In fact, as we have confirmed it by computation experiments, the formula (6) is better than both uniform one $1/|S|$ and proportional one $\sum_{i=1}^{n} v_i^s / \sum_{t\in S} \sum_{i=1}^{n} v_i^t$.

Remark. Since we would like to obtain rather small upper approximation of the optimal value of KP (5), it is preferable that the optimal value of KP (5) own is as small as possible. In other words, we would like to determine a multiplier vector so that the KP (5) has an optimal value as small as possible. In fact among the three types of multiplier vectors, the proportional one is worst. This implies that, on each scenario s, the larger $\sum_{i=1}^{n} v_i^s$ is, the smaller μ_s should be.

As it will be cleared afterwards, upper bound z_{SL} is terrible (see also Table 3 presented subsequently). In the next subsection we attempt to exploit an iterative processing to find an appropriate multiplier vector like subgradient method does [2].

3.4 Surrogation multiplier and iteration

In this subsection, inspired by the result in the preceding subsection, we discuss another multiplier vector which is obtained by exploiting an iterative processing.

First we have to assign initial value to each $\mu_s\,(s \in S)$ which is supplied to scenario s in the KP (5). Here we will denote the value of the result of LP relaxation applied to MNK assumed that there exists only one scenario s, which can be regarded as KP, by $\bar{Z}_D(\mu_s)$. By way of trial we will determine an initial μ_s as the inverse of $\bar{Z}_D(\mu_s)$ for each $s \in S$. This is because, as mentioned in the preceding remark, to hold the optimal value of the KP (5) as small as possible, it would be natural to think that the larger $\bar{Z}_D(\mu_s)$ is, the smaller μ_s should be. Thus we have new upper bound by the following procedure:

foreach $s \in S$ $\{\mu_s := 1/\bar{Z}_D(\mu_s)\}$;
Normalize $(\mu_s)_{s\in S}$;
UB := result of LP relaxation applied to the KP (5) with $(\mu_s)_{s\in S}$;
for (;;) {
    foreach $s \in S$ $\{\mu_s := (\mu_s)^2\}$ /* bias */
    Normalize $(\mu_s)_{s\in S}$;
    tmpUB := result of LP relaxation applied to the KP (5) with $(\mu_s)_{s\in S}$;
    if (tmpUB < UB)
        UB := tmpUB;
    else
        break; /* exit infinite loop */
}
Output UB;

After obtaining the UB, each multiplier is multiplied by itself. This operation gives a bias to each multiplier so that the difference between any pair of multipliers will be widen gradually. After that the set of multipliers is normalized to satisfy the constraint of the KP (5), that is, $\sum_{s\in S} \mu_s = 1$. The processing continues until UB does not improve.

Here we present computation experiments in Table 3. Each line including z_{EG}, z_{SL} and z_{SiL} expresses the average lower or upper bound of 100 data instances. Columns for $(z_U - z_L)/z_U$ express the gap between z_U upper and z_L lower bounds proposed in [5] as is for reference. In Table 3, upper bound z_{SiL} gained by the iterative processing is a little better than z_{SL}. However it is quite far from the result in [5], and there exists large gap between z_{EG} and z_{SiL}.

By additional computation experiments, we have confirmed that the gap of our proposed bounds is still large even if n becomes large. When n is even large, it is empirically well-known that LP relaxation gives sharp upper bound for KP. Furthermore the tightness of surrogate relaxation technique for MNK is theoretically and experimentally warrantied in [5]. Therefore this gap should be owing to not LP relaxation technique but the determination method for multiplier vector.

3.5 Lower bound again

The upper and lower bounds proposed in [5] are simultaneously produced by the procedure named *SurrogateBounds*. In the procedure, lower bound is obtained by using the solution which

Table 3:

δ	n	$\lvert S\rvert$	m	z_{EG}	z_{SL}	z_{SiL}	$\frac{z_{\mathrm{SiL}}-z_{\mathrm{EG}}}{z_{\mathrm{SiL}}}$	$\frac{z_U-z_L}{z_U}$
0.9	60	10	2	2004.94	2414.58	2206.73	0.0914	0.0113
			3	1616.64	1992.15	1854.25	0.1281	0.0186
			4	1367.49	1732.70	1601.82	0.1462	0.0167
		20	2	1965.95	2398.54	2139.40	0.0810	0.0105
			3	1579.89	1997.19	1790.90	0.1178	0.0192
			4	1304.37	1702.63	1522.37	0.1431	0.0266
		30	2	1942.33	2403.52	2110.53	0.0796	0.0184
			3	1541.55	1982.09	1739.68	0.1138	0.0167
			4	1310.72	1728.20	1516.81	0.1358	0.0207

z_{EG}: enhanced greedy (lower bound)
z_{SL}: surrogate & LP relaxation (upper bound)
z_{SiL}: surrogate & iterative LP relaxation (upper bound)
$(z_U - z_L)/z_U$: as is in [5]

gives an optimal value of the KP (5) in every iteration. Similar to this idea, we will attempt to rewrite the procedure in the preceding subsection as follows:

Bounds(Out: LB, UB)
  foreach $s \in S$ $\{\mu_s := 1/\bar{Z}_D(\mu_s)\}$
  Normalize $(\mu_s)_{s\in S}$;
  Apply LP relaxation to the KP (5) with $(\mu_s)_{s\in S}$,
    and obtain UB and solution x^* which gives UB;
  LB := $\min_{s\in S} \sum_{i\in N} v_i^s x_i^*$;
  for (;;) {
    foreach $s \in S$ $\{\mu_s := (\mu_s)^2\}$ /* bias */
    Normalize $(\mu_s)_{s\in S}$;
    Apply LP relaxation to the KP (5) with $(\mu_s)_{s\in S}$,
      and obtain tmpUB and solution x^* which gives tmpUB;
    tmpLB := $\min_{s\in S} \sum_{i\in N} v_i^s x_i^*$;
    if (tmpUB < UB)
      UB := tmpUB;
    if (tmpLB > LB)
      LB := tmpLB;
    if (neither UB nor LB improves)
      break; /* exit infinite loop and return */
  }

It would be convenient that we will provide a subroutine which receives a multiplier vector supplied to the KP (5) and applies LP relaxation to it, which returns caller UB and solution x^* which gives UB. Note that the last taken item i_k may not be included in x^* if it is used partially, that is, $b - \sum_{j=1}^{k-1} a_{i_j} < a_{i_k}$. This is to ensure the feasibility of the solution x^* which also gives LB.

In addition we attempt to use other two initial multiplier vectors in place of $(1/\bar{Z}_D(\mu_s))_{s\in S}$: one is the formula (6) in Subsection 3.3, the other is the one which is inversely proportional to

efficiency sum as follows:

$$\mu_s = \frac{\dfrac{1}{\sum_{i=1}^{n} v_i^s/a_i}}{\sum_{t\in S} \dfrac{1}{\sum_{i=1}^{n} v_i^t/a_i}}. \tag{7}$$

Then, the procedure *Bounds* should be modified so that it can receive an initial multiplier vector. Hereafter, we assume that the modification has been done. By computation experiments, the usual initial multiplier vector of $(1/\bar{Z}_D(\mu_s))_{s\in S}$ is best as to not only upper but also lower bound among the three in overwhelming cases, however, it is not always so. Therefore we adopt all of these three types of initial multiplier vectors. Moreover we also adopt another candidate, which is minimal $\bar{Z}_D(\mu_s)$ gained under all $s \in S$.

Remark. The multiplier vector $(\mu_s)_{s\in S}$ that $\mu_s = 1$ and other $|S| - 1$ multipliers are zero satisfies the restriction of the problem (5), that is, $\mu_s \geq 0$ for all $s \in S$ and $\sum_{s\in S} \mu_s = 1$. Thus obtained $\bar{Z}_D(\mu_s)$ indeed gives an alternative upper bound. In a word for each $s \in S$, MNK assumed that there exists only one scenario s is a relaxation. We will denote it by $Z_D(\mu_s)$, which is formulated as follows:

$$Z_D(\mu_s) = \max_x \left\{ \sum_{i=1}^{n} v_i^s x_i \;\middle|\; \sum_{i=1}^{n} a_i x_i \leq b;\ x_i \in \{0,1\},\ i = 1,\ldots,n \right\} \tag{8}$$

Consequently we use four candidates to obtain upper bound. By our computation experiments, minimal $\bar{Z}_D(\mu_s)$ gained under all $s \in S$ is not so tight, however, it sometimes produces good one rather than the procedure *Bounds* called with any of three types of initial multiplier vectors. Moreover this candidate is incidentally obtained in the processing of computing $(1/\bar{Z}_D(\mu_s))_{s\in S}$. Hence it is a pity to discard the candidate.

On the other hand on lower bound, we have already proposed z_{EG} which is maximal of two greedy heuristics introduced in Subsection 2.2. As mentioned just previously, the procedure *Bounds* called with $(1/\bar{Z}_D(\mu_s))_{s\in S}$ gives almost best among the five, however, it is not always so. Thus we will exploit all of the five candidates for lower bound.

Here we present computation experiments for the tightness of our proposed bounds in Table 4. Each line including z_{XL} and z_{XU} expresses the average lower or upper bound of 100 data instances respectively. Columns for $(z_U - z_L)/z_U$ are as is in Table 3 for reference, which will be denoted by the symbol Δ_2. As we can see in Table 4, the gap between our proposed bounds denoted by the symbol Δ_1 is roughly two or three times worse than Δ_2.

An additional remark is that the reference [2] includes suggestive words: "Two properties are important in evaluating a relaxation: the sharpness of the bounds produced and the amount of computation required to obtain these bounds. Usually selecting a relaxation involves a tradeoff between these two properties; sharper bounds require more time to compute." The bounds proposed in [5] are to be sure very tight, whereas it would seem that it takes too many costs to obtain them, because the procedure which computes the bounds exactly solves KP in every iteration. On the other hand our proposed bounds can be obtained immediately, whereas they are two or three times loose than those in [5]. It would be thus interesting to see which is better to solve MNK.

4 An algorithm for the MNK problem

In this section we develop an algorithm to solve the MNK problem. To solve the combinatorial optimization problem, branch-and-bound is so popular. In what follows we implement our

Table 4: Our bounds

δ	n	$\lvert S\rvert$	m	z_{XL}	z_{XU}	$\frac{z_{\mathrm{XU}}-z_{\mathrm{XL}}}{z_{\mathrm{XU}}}$	$\frac{z_U-z_L}{z_U}$	Δ_1/Δ_2
0.3	60	10	2	2299.78	2315.79	0.0069	0.0042	1.64
			3	1893.86	1915.67	0.0114	0.0026	4.38
			4	1608.48	1631.94	0.0144	0.0047	3.06
		20	2	2266.83	2282.68	0.0069	0.0035	1.97
			3	1853.29	1877.40	0.0128	0.0060	2.13
			4	1588.95	1614.07	0.0156	0.0041	3.80
		30	2	2270.04	2288.82	0.0082	0.0049	1.67
			3	1825.42	1850.17	0.0134	0.0059	2.27
			4	1585.21	1608.86	0.0147	0.0086	1.70
0.6	60	10	2	2252.36	2283.72	0.0137	0.0066	2.07
			3	1783.21	1831.59	0.0264	0.0106	2.49
			4	1536.77	1583.33	0.0294	0.0044	6.68
		20	2	2185.38	2220.64	0.0159	0.0091	1.74
			3	1787.21	1842.35	0.0299	0.0095	3.14
			4	1487.19	1543.26	0.0363	0.0104	3.49
		30	2	2146.95	2183.68	0.0168	0.0044	3.81
			3	1737.64	1797.26	0.0332	0.0141	2.35
			4	1480.66	1541.59	0.0395	0.0188	2.10
0.9	60	10	2	2154.57	2211.20	0.0256	0.0113	2.26
			3	1786.83	1865.45	0.0421	0.0186	2.26
			4	1502.88	1592.31	0.0562	0.0167	3.36
		20	2	2091.57	2163.35	0.0332	0.0105	3.16
			3	1672.70	1769.75	0.0548	0.0192	2.85
			4	1428.60	1520.98	0.0607	0.0266	2.28
		30	2	2018.60	2095.58	0.0367	0.0184	1.99
			3	1602.45	1702.71	0.0589	0.0167	3.52
			4	1401.91	1511.84	0.0727	0.0207	3.51

z_{XL}: lower bound by the maximal of the five candidates
z_{XU}: upper bound by the minimal of the four candidates
Δ_1 : $(z_{\mathrm{XU}} - z_{\mathrm{XL}})/z_{\mathrm{XU}}$
Δ_2 : $(z_U - z_L)/z_U$, as is in [5]

algorithm based on the branch-and-bound.

To begin with, we obtain initial lower bound among the five candidates: z_{EG} and the procedure *Bounds* called with three types of initial multiplier vectors introduced in the preceding section. The obtained initial lower bound is set to initial incumbent value. Hereafter we will denote the incumbent value by z^*. The incumbent value z^* would be replaced with larger value given by a feasible solution which would be found during exploring search-tree. Also initial upper bound is computed simultaneously, which is the best one among the four candidates. If the upper bound is corresponding to z^* then program is terminated, because we have obtained optimal value z^*. In passing, on the implementation, the solution which gives z^* should be always memorized.

The search-tree is a binary tree and each node has two children (subproblems) prescribed by fixing one item whether we take it or not. We will call the item *branching variable*. On each node, branching variable is determined dynamically as $i \in N_2$ which gives maximal of $\sum_{s\in S} \mu_s^* v_i^s / a_i$ following to [5], where $(\mu_s^*)_{s\in S}$ indicates multiplier vector obtained by computing upper bound at each node. More precisely, the procedure computing upper bound should return multiplier vector which prescribes the KP which gives output of UB to the caller. Rough algorithmic sketch of this processing is presented subsequently.

As soon as we visit a node the following two processings are performed. If the set N_2 becomes empty consequently or has been already empty, then we do not explore search-tree more deeply. In other words no subproblem of which parent is the visited node is spawned and the node own is discarded. Note that the case where N_2 is already empty is also included in each case respectively.

- If we can take all items in N_2, that is, $\sum_{i\in N_2} a_i \leq b - \sum_{i\in N_1} a_i$, then all items in N_2 are moved to N_1.
- We will sweep out an item j in N_2 to N_0, provided $a_j > b - \sum_{i\in N_1} a_i$.

Before discarding the node in each case, $\min_{s\in S} \sum_{i\in N_1} v_i^s$ is computed, and compared with z^*. The greater is left as new z^*.

On each node of search-tree, we should consider a subproblem of given MNK, which is prescribed by each node as:

$$z'_{\mathrm{MNK}} = \max_{I\subset N_2} \min_{s\in S} \left\{ \sum_{i\in I\cup N_1} v_i^s \;\middle|\; \sum_{i\in I\cup N_1} a_i \leq b \right\}. \tag{9}$$

Thus the triplet of each node of $\{N_0, N_1, N_2\}$ prescribes the subproblem concerned with the node. For instance, the triplet of $\{\emptyset, \emptyset, N\}$ is concerned with top node in which all items have not yet been fixed. Also the surrogate relaxation of the problem (9) is formulated as follows:

$$Z'_D(\mu) = \max_{I\subset N_2} \left\{ \sum_{i\in I\cup N_1} v_i(\mu) \;\middle|\; \sum_{i\in I\cup N_1} a_i \leq b,\ v_i(\mu) = \sum_{s\in S} \mu_s v_i^s \right\}, \tag{10}$$

where $\mu_s \geq 0$ for any $s \in S$ and $\sum_{s\in S} \mu_s = 1$.

On computing upper bound on every node except top node, we also exploit the procedure *Bounds* with three types of initial multiplier vectors. The termination condition of the infinite loop in the procedure is however modified so that it is concerned with the improvement of upper bound only, so the improvement of lower bound is not concerned as follows:

Iter_Proc(In: $(\mu_s)_{s\in S}$, Out: LB, UB, $(\mu_s^*)_{s\in S}$)
    Normalize received $(\mu_s)_{s\in S}$; /* three types of $(\mu_s)_{s\in S}$ */
    Apply LP relaxation to the KP (10) with $(\mu_s)_{s\in S}$,
        and obtain UB and solution x^* which gives UB;
    Save $(\mu_s)_{s\in S}$ as $(\mu_s^*)_{s\in S}$;
    LB := $\min_{s\in S} \sum_{i\in N} v_i^s x_i^*$;
    for (;;) {
        foreach $s \in S$ $\{\mu_s := (\mu_s)^2\}$ /* bias */
        Normalize $(\mu_s)_{s\in S}$;
        Apply LP relaxation to the KP (10) with $(\mu_s)_{s\in S}$,
            and obtain tmpUB and solution x^* which gives tmpUB;
        tmpLB := $\min_{s\in S} \sum_{i\in N} v_i^s x_i^*$;
        if (tmpLB > LB)
            LB := tmpLB;
        if (tmpUB < UB) {
            UB := tmpUB;
            Save $(\mu_s)_{s\in S}$ as $(\mu_s^*)_{s\in S}$;
        } **else**
            break; /* exit infinite loop and return */
    }

Our main purpose of this procedure is to obtain upper bound quickly. In fact by our computation experiments, it seems that the processing of obtaining lower bound and trying to improve z^* during exploring search-tree is not so effective. Therefore we have concentrated our attention only upon the improvement of upper bound during exploring search-tree. In other words, obtaining lower bound in the procedure *Iter_Proc* is a mere wish, so it might not be necessary.

The output $(\mu_s^*)_{s\in S}$ of the *Iter_Proc* is exploited to determine branching variable as mentioned previously. Although we should modify the procedure *Bounds* so that it also returns the multiplier vector for the branching variable on top node, we here apply the *Iter_Proc* to the top node in order to obtain it for a while. In fact this processing will become unnecessary because of exploiting new branching variable.

A remark is that the initial multiplier vector produced by $(1/\bar{Z}_D(\mu_s))_{s\in S}$ is determined so that a triplet of node is taken into account. On the other hand the rest of two initial multiplier vectors, the formulae (6) and (7), are always constant respectively. In addition as we have mentioned it for the *Bounds*, the solution x^* appeared in the procedure should not include an item which is used partially.

The remainder of this section is devoted to the improvement of our proposed algorithm. As things turned out, we would improve our procedure for the bounds in some cases.

4.1 Search strategy

In this subsection we will implement our algorithm based on two search strategies respectively and compare the performance of them: one is classical depth-first search, the other is best-first search adopted in [5].

In depth-first search, on each visited node, upper bound of the subproblem which is prescribed by the node is computed after performing the previous two processings as to N_2. If obtained upper bound is less than or equal to incumbent value z^*, then the node is discarded; otherwise we determine branching variable and spawn two subproblems of which parent is cur-

rently visited node. On visiting two children, we first try to take branching variable i, that is, after exploring sub-tree with $x_i = 1$, we explore sub-tree with $x_i = 0$.

On the other hand, in best-first search, active nodes (subproblems) are stored in a list. The nodes in the list are sorted according to the upper bound of the subproblem prescribed by each node in a nonincreasing order. Therefore the upper bound for each node should be computed before adding the node to the list, because it is prerequisite to determine a position where the node should be placed in the list. By this, a node which should be expanded next can be always taken from the head of the list with no condition. After taking a node from the head of the list, the upper bound of the node is compared with z^* immediately. After that, the previous two processings as to N_2 are performed on and two children are spawned if necessary.

Computation experiments are shown in Table 5. Each column expresses the average of 100 data instances, so each strategy is not applied to same 100 instances. Columns for BBN indicate the average total number of visited nodes during exploring search-tree. The BBN includes the number of nodes which are not expanded. Also a conspicuous anomalous instance in each column is presented in the brackets on the column for BBN as '(worst).'

Unfortunately we can not conclude that which search strategy is better as far as exploiting our bounds. Under our observation, it seems that BBN as for each algorithm is highly dependent on the instance of MNK respectively. For instance, each has solved some instances with BBN under 1,000, whereas solved some instances with BBN over 10,000. In fact this computation result is due to the branching variable. We will introduce new branching variable in the next subsection, which really improves our algorithm.

Table 5: Two search strategies

δ	n	$\|S\|$	m	depth-first BBN(worst)	best-first BBN(worst)
0.9	60	10	2	1445.5(11,897)	1368.8(12,349)
			3	2655.5(26,977)	2428.8(16,861)
			4	2171.4(14,493)	2238.0(21,099)
		20	2	2109.5(13,459)	2542.9(44,195)
			3	4510.3(33,913)	4712.3(50,073)
			4	3602.0(23,809)	5053.2(48,671)
		30	2	4066.8(35,287)	3656.7(59,545)
			3	6818.2(85,545)	6126.6(51,427)
			4	4906.5(33,945)	5508.9(49,343)

4.2 Branching variable

It is important to take an appropriate branching variable in order to reduce BBN. It is however difficult to find the most valuable item on MNK, because there exists no general measure for the efficiency of an item. The discussion in Section 2 also implies this fact. If there exists such a general measure like as classical KP, it should be useful to determine branching variable. In this subsection we exploit another branching variable, which not only improves our proposed algorithm but also has a merit on the point of view of computation time.

While it is not mentioned why item $i \in N_2$ which gives maximal of $\sum_{s \in S} \mu_s^* v_i^s / a_i$ is adopted as branching variable in [5], it would be appropriate from the point of view of common sense on KP, assuming that the KP (5) is an approximation of given MNK. However we here exploit the alternative which gives maximal of $\sum_{s \in S} v_i^s$. Namely, our suggestion is that the larger $\sum_{s \in S} v_i^s$ is, the larger the probability that item i would be included in an optimal solution is.

Computation experiments of exploiting this decision are presented in Table 6. Each column expresses the average of 100 data instances, so each strategy is not applied to same 100 instances. As we can see in Table 6, our algorithms have been quite improved. The reason would be that the KP (5) which is the approximation of given MNK is not precise. Hence it has caused weird choice of branching variable.

Also in Table 6, best-first is better than depth-first. This is suitable to our intuition very well. In a word best-first search is better than depth-first search on reducing BBN. In addition BBN in the worst case presented in Table 6 is well-bounded and conspicuous anomalous instances have not found. Therefore we hereafter adopt best-first search with the new decision of branching variable. Moreover, by this, it is no more necessary for the *Iter_Proc* to return $(\mu_s^*)_{s\in S}$ which prescribes the KP giving output of UB to caller.

Table 6: New decision of branching variable

δ	n	$\|S\|$	m	depth-first BBN(worst)	best-first BBN(worst)
0.9	60	10	2	1417.2(20,609)	888.7(3,711)
			3	1830.2(13,633)	976.8(4,759)
			4	1700.7(8,035)	915.4(4,043)
		20	2	1745.6(10,291)	876.1(4,149)
			3	2529.8(11,251)	1245.8(6,219)
			4	2789.2(19,893)	1200.0(10,175)
		30	2	2215.6(28,297)	1251.0(15,837)
			3	3072.6(16,557)	1885.6(8,963)
			4	3075.0(13,997)	1571.7(18,957)

Furthermore, the new decision of branching variable has one merit as we have mentioned it at the beginning of this subsection. Namely, we can determine the order of an item which should be taken as branching variable before exploring search-tree. On implementation, all items are sorted in a nonincreasing order of $\sum_{s\in S} v_i^s$ and are stored in a list as N_2. Then, branching variable can be always taken from the head of the list with no condition.

On the other hand the decision of branching variable can be regarded as an analogy of the value-ordered greedy heuristic. Since there exist other three greedy heuristics for MNK, we can discuss at least more three decisions as the analogy of each greedy heuristic respectively. Then, we have performed computation experiments with sorting all items beforehand according to the three analogies respectively: Weight-ordered decision is taking item i which minimizes a_i. Density-ordered one is taking which maximizes $\sum_{s\in S} v_i^s/a_i$. Total-value one is which maximizes $\sum_{s\in S} v_i^s \lfloor b/a_i \rfloor$. Consequently usual value-ordered is most effective among the four.

4.3 Temporary implementation

In this subsection we present extensive computation experiments of our proposed algorithm so far with several types of data instances. Repeating that it exploits best-first search and to the top node we apply the procedure *Bounds* and to the rest of all nodes we apply the procedure *Iter_Proc*. It also includes the processing of sorting of all items before exploring search-tree for the branching variable.

Computation experiments are in Table 7. In the table, each column expresses the average of 100 data instances. Computation time is expressed in seconds, which is shown in the columns for CPU. Also time to sort all given items is included in CPU. Reference machine, throughout

this report, is SPARCstation-20[†]. The algorithm has been implemented in C and program is complied by gcc with –O3 option. On the other hand the columns for Yu are as is in [5] for reference, where the reference machine is IBM3090 and the algorithm was implemented in PASCAL. Needless to say, exact comparison of computation time (CPU) is not so meaningful.

Table 7 implies that our algorithm is good at the case where δ is small, which could be easily guessed from the result in Table 4. Also our algorithm shows bad performance as $|S|$ becomes large. Since it seems that our proposed bounds are not so sensitive to $|S|$ by the result in Table 4, the defect would be owing to the increase of computation time as $|S|$ has grown (For instance, the initial multiplier vector $(1/\bar{Z}_D(\mu_s))_{s\in S}$ which is supplied to *Iter_Proc*). Consequently when $|S|$ is large, our processing for the bounds is not so light. The difference between two algorithms as to BBN in Table 7 apparently exhibits how stable and tight the bounds proposed in [5] are.

A remark is that BBN tends to be best in the case where $m = 2$ under any set of δ, n and $|S|$ on our algorithm. It is a strange behavior because in the case where $m = 2$, total number of feasible solutions is greatest among the three m in general. Therefore BBN (also CPU) should be smallest in the case where $m = 4$. The algorithm in [5] shows such a behavior, whereas ours shows the best behavior in the case where $m = 2$.

An additional remark is that under our observation, when $\delta = 0.3$, our initial lower bound is frequently corresponding to an optimal value, however, our algorithm solves such an instance with BBN over 100. It exhibits that our task which we should address is no doubt how we obtain sharp upper bound by some means or other. To cope with it, the substantial issue is how we determine appropriate multiplier vector which prescribes KP (5) as the relaxation of given MNK and its subproblems, in other words, how we find a precise approximation of MNK.

4.4 Step size

In order to improve the precision of the approximation of MNK, we will exploit new step size $\mu_s := (\mu_s)^n \, (1 < n < 2)$ in two iterative procedures: *Bounds* and *Iter_Proc*. Since the new step size is shorter than the old, the difference between any pair of multipliers will be widen with more short step.

Computation experiments for this trial are in Table 8. Each line expresses the average of 100 data instances. The meaning of columns for z_{XL} and z_{XU} are same as those in Table 4. As we can see in Table 8 the smaller the step is, the larger the computation time, which is not preferable. In addition it seems that the small step contributes nothing to the improvement of BBN. Therefore we should not adopt new step size at least for the *Iter_Proc*.

On the other hand, $\mu_s := \mu_s \sqrt[4]{\mu_s}$ is most efficient for the improvement of the Δ_1 in Table 8, however, it is not always so. For instance in the case where $|S| = 10$ and $m = 3$, $\mu_s := \mu_s \sqrt{\mu_s}$ gives best among all step sizes. Also in the case where $(|S|, m) = (20, 4)$ or $(30, 3)$, $\mu_s := \mu_s \sqrt[8]{\mu_s}$ gives best.

Since the procedure *Bounds* is called only once for each initial multiplier vector, it does not matter that it costs a little. Therefore we will adopt new step size for the *Bounds*: First we use ordinary $\mu_s := (\mu_s)^2$ and continue so that $\mu_s := \mu_s \sqrt[n]{\mu_s}$ where $n = 2^m \, (m \geq 1, \text{ integer})$ for the direction that m is increasing. If lower or upper bound produced in performing dynamical step size would improve, then each currently stored bound should be replaced with the improved one respectively. The processing will continue until the gap between stored upper and lower bounds does not improve in three times contiguously. This means that we would like to perform the processing at least for $m = 1, 2$ and 3. The effect of dynamical step size will be shown in the next subsection by the *Bounds* which includes this feature.

[†]within SuperSPARC II (75 MHz). The numerical performance is 125.8 SPECint92/121.2 SPECfp92.

Table 7: Comparison of two algorithms

δ	n	$\|S\|$	m	by Yu* BBN	by Yu* CPU (sec.)	by Iida** BBN(worst)	by Iida** CPU (sec.)
0.3	60	10	2	43.0	227.5	245.2(801)	4.1
			3	36.4	200.8	319.8(1,059)	5.0
			4	29.6	120.1	264.8(1,297)	3.9
		20	2	42.4	217.9	240.4(715)	7.1
			3	35.4	234.3	331.6(1,599)	9.1
			4	29.8	200.7	310.0(1,311)	8.0
		30	2	40.8	221.5	247.7(1,375)	10.9
			3	36.0	229.2	348.1(2,389)	15.0
			4	29.8	190.2	307.4(1,509)	12.5
0.6	60	10	2	43.7	231.3	353.5(1,493)	5.7
			3	36.2	270.8	541.9(5,643)	8.0
			4	30.0	204.9	437.9(3,147)	6.1
		20	2	42.2	245.5	483.6(1,989)	13.6
			3	35.0	243.3	603.5(2,933)	15.9
			4	33.0	212.1	492.0(2,167)	12.3
		30	2	43.2	241.3	436.7(2,559)	17.8
			3	35.8	227.6	598.6(2,613)	24.1
			4	31.6	191.2	500.0(2,005)	19.4
0.9	60	10	2	44.8	334.0	708.4(2,247)	10.2
			3	38.0	322.4	1097.8(7,677)	14.6
			4	31.0	253.2	1005.8(5,355)	12.7
		20	2	44.2	262.4	1069.1(4,311)	28.4
			3	36.4	246.9	1546.7(9,971)	37.7
			4	31.4	207.4	1451.5(7,845)	34.3
		30	2	44.8	252.3	1153.9(4,945)	45.8
			3	38.0	234.9	1791.9(8,815)	73.2
			4	30.0	189.1	1318.5(5,603)	47.7
		50	2	—	—	1405.3(27,567)	109.2
			3	—	—	2326.9(21,793)	162.8
			4	—	—	1664.2(8,159)	124.0
	70	50	2	—	—	2301.1(12,663)	202.4
			3	—	—	3378.8(20,389)	292.9
			4	—	—	3558.9(22,037)	298.8
	80	50	2	—	—	5869.3(79,845)	627.6
			3	—	—	7260.9(54,595)	751.6
			4	—	—	5786.0(39,269)	606.1

$*$: solved on IBM3090

$**$: solved on SPARCstation-20 with gcc –O3

Table 8: New step size

$\mu_s :=$	δ	n	$\lvert S\rvert$	m	BBN(worst)	CPU (sec.)	z_{XL}	z_{XU}	Δ_1
$\mu_s \sqrt[4]{(\mu_s)^3}$	0.9	60	10	2	741.74(3,429)	11.8	2164.57	2222.44	0.0260
				3	987.54(6,055)	14.8	1748.43	1832.19	0.0457
				4	1030.92(7,147)	14.8	1500.91	1588.90	0.0554
			20	2	811.68(6,903)	23.3	2058.40	2123.18	0.0305
				3	1310.20(7,871)	35.7	1664.30	1757.03	0.0528
				4	1064.60(7,073)	27.5	1423.81	1520.61	0.0637
			30	2	1236.10(4,965)	54.2	2027.82	2105.39	0.0368
				3	1893.24(25,851)	78.4	1631.82	1735.99	0.0600
				4	1345.92(9,375)	53.4	1376.22	1479.13	0.0696
$\mu_s \sqrt{\mu_s}$	0.9	60	10	2	762.14(5,455)	14.5	2156.43	2213.04	0.0256
				3	897.18(3,391)	17.0	1769.29	1840.67	0.0388
				4	873.04(3,299)	17.6	1485.05	1575.27	0.0573
			20	2	920.58(5,797)	31.0	2085.85	2148.59	0.0292
				3	1381.56(11,243)	43.3	1685.67	1775.80	0.0508
				4	983.56(4,607)	29.8	1414.72	1508.85	0.0624
			30	2	914.50(3,743)	45.9	2048.34	2117.40	0.0326
				3	1879.28(20,451)	87.6	1616.50	1718.49	0.0593
				4	1608.70(7,759)	72.4	1402.36	1508.26	0.0702
$\mu_s \sqrt[4]{\mu_s}$	0.9	60	10	2	706.44(6,011)	19.2	2175.59	2225.49	0.0224
				3	1037.98(4,389)	25.1	1744.74	1818.60	0.0406
				4	920.78(6,917)	21.1	1536.13	1611.67	0.0469
			20	2	834.24(9,853)	40.6	2086.13	2139.30	0.0249
				3	1001.24(5,423)	45.6	1696.10	1779.58	0.0469
				4	1174.74(5,239)	50.3	1427.52	1524.14	0.0634
			30	2	1235.82(11,267)	83.6	2036.98	2100.35	0.0302
				3	1633.88(19,767)	106.5	1648.87	1745.42	0.0553
				4	1559.96(6,757)	96.4	1380.22	1482.62	0.0691
$\mu_s \sqrt[8]{\mu_s}$	0.9	60	10	2	1375.92(12,093)	38.9	2168.91	2228.85	0.0269
				3	1517.56(13,743)	40.9	1771.07	1852.52	0.0440
				4	1695.08(10,243)	34.8	1496.43	1598.06	0.0636
			20	2	1166.26(6,411)	81.6	2051.96	2114.62	0.0296
				3	1732.54(18,221)	103.4	1685.51	1775.75	0.0508
				4	1196.04(9,813)	69.6	1423.61	1516.34	0.0612
			30	2	1128.78(5,879)	115.1	2046.03	2110.84	0.0307
				3	1149.04(7,871)	112.1	1637.05	1719.91	0.0482
				4	1619.18(10,537)	138.5	1390.62	1496.61	0.0708

Δ_1: $(z_{\mathrm{XU}} - z_{\mathrm{XL}})/z_{\mathrm{XU}}$

4.5 Other initial multiplier vectors

In the former half of this subsection, we attempt to exploit other three types of initial multiplier vectors for the *Bounds* which is performed on top node. In the latter half we also attempt to exploit new initial multiplier vectors for the *Iter_Proc* which is performed on the rest of all nodes (We call the node intermediate node).

4.5.1 On top node

As we have mentioned previously, $1/\bar{Z}_D(\mu_s)$ is most effective as to initial multiplier vector supplied to the *Bounds*. In this subsubsection we discuss other initial multiplier vectors which are similar to $1/\bar{Z}_D(\mu_s)$ respectively.

New three types of initial multiplier vectors to each scenario s are as follows, which are based on the greedy heuristics for IKP respectively:

1. The inverse of the result of density-ordered greedy heuristic applied to the KP (8).
2. In the previous clause, using total-value in place of density-ordered.
3. In the first clause, using value-ordered in place of density-ordered.

Then, we will call the *Bounds* six times with different types of initial multiplier vectors severally. Consequently we have exploited eight candidates for the initial lower bound and seven candidates for the initial upper bound respectively.

Moreover in the *Bounds*, we attempt to reuse a solution which will give output of UB. Assuming that solution x^* gives final output of UB in the *Bounds*, we use the inverse of $\sum_{i \in N} v_i^s x_i^*$ as an initial multiplier to each scenario s and perform the iterative processing again. If the upper bound obtained by the new initial multiplier improves, then the improved one will be returned to caller. On the other hand as to a solution which gives lower bound LB in the *Bounds*, we also perform the same processing except using $\sum_{i \in N} v_i^s x_i^*$ as is, not taking the inverse of it.

Computation experiments are presented in Table 9. In the table each line expresses the average of 100 data instances, and the best result as to Δ_1/Δ_2 among many trials. Also the procedure *Bounds* has adopted dynamical step size illustrated in Subsection 4.4. In Table 9, our proposed bounds have slightly improved as to Δ_1/Δ_2 compared with Table 4 as we anticipated it. However it can not yet reach the result in [5].

4.5.2 On intermediate node

So far we have exploited three types of initial multiplier vectors for the procedure *Iter_Proc*: $1/\bar{Z}_D(\mu_s)$, formulae (6) and (7). In addition $\min_{s \in S} \bar{Z}_D(\mu_s)$ has been a candidate for upper bound. In this subsubsection we will introduce other initial multiplier vectors for the *Iter_Proc*, and explore several combinations of all candidates involving those induced by the new initial multiplier vectors respectively. In what follows, at the beginning of each three paragraph, we will mention what is the difference from the default of four candidates in each trial. In addition in this subsubsection the procedure *Bounds* is always as is improved in Subsubsection 4.5.1. Also all computation experiments in this subsubsection express the average of 100 data instances.

First, we discard usual four candidates and exploit only one new candidate, which is the output of *Iter_Proc* called with the first clause (density-ordered) in the previous new three types of initial multiplier vectors. Computation experiments are in Table 10. In this trial computation time improves when $\delta = 0.9$, because we call *Iter_Proc* only once on every node. However this

Table 9: Eight and seven candidates with dynamical step size

δ	n	$\|S\|$	m	z_{XL}	z_{XU}	Δ_1	Δ_2	Δ_1/Δ_2
0.3	60	10	2	2287.92	2302.08	0.0062	0.0042	1.47
			3	1879.27	1898.92	0.0103	0.0026	3.96
			4	1633.37	1652.93	0.0118	0.0047	2.51
		20	2	2274.73	2290.29	0.0068	0.0035	1.94
			3	1866.55	1885.86	0.0102	0.0060	1.70
			4	1590.65	1612.85	0.0138	0.0041	3.36
		30	2	2306.61	2322.62	0.0069	0.0049	1.40
			3	1846.00	1867.11	0.0113	0.0059	1.91
			4	1596.92	1620.79	0.0147	0.0086	1.70
0.6	60	10	2	2262.04	2286.79	0.0108	0.0066	1.63
			3	1800.77	1834.82	0.0186	0.0106	1.75
			4	1561.05	1599.43	0.0240	0.0044	5.45
		20	2	2161.76	2188.79	0.0123	0.0091	1.35
			3	1785.16	1826.16	0.0225	0.0095	2.36
			4	1516.50	1559.18	0.0274	0.0104	2.63
		30	2	2149.50	2175.31	0.0119	0.0044	2.70
			3	1786.79	1827.93	0.0225	0.0141	1.59
			4	1467.07	1510.59	0.0288	0.0188	1.53
0.9	60	10	2	2182.74	2222.15	0.0177	0.0113	1.56
			3	1770.81	1824.93	0.0297	0.0186	1.59
			4	1508.26	1570.54	0.0397	0.0167	2.37
		20	2	2087.02	2132.09	0.0211	0.0105	2.00
			3	1707.06	1775.26	0.0384	0.0192	2.00
			4	1465.38	1537.90	0.0472	0.0266	1.77
		30	2	2085.22	2138.21	0.0248	0.0184	1.34
			3	1652.41	1728.39	0.0440	0.0167	2.63
			4	1410.48	1487.87	0.0520	0.0207	2.51

z_{XL}: lower bound by the maximal of the eight candidates
z_{XU}: upper bound by the minimal of the seven candidates
Δ_1 : $(z_{\mathrm{XU}} - z_{\mathrm{XL}})/z_{\mathrm{XU}}$
Δ_2 : $(z_U - z_L)/z_U$, as is in [5]

Table 10: Density-ordered only

δ	n	$\|S\|$	m	BBN(worst)	CPU (sec.)
0.3	60	10	2	2147.14(78,823)	10.2
0.9	60	10	2	827.50(4,705)	7.2
			3	1040.84(6,635)	8.0
			4	904.10(4,397)	6.4
		20	2	1104.62(11,267)	16.0
			3	1773.70(12,041)	22.6
			4	1453.30(6,109)	17.5
		30	2	1085.32(7,299)	22.3
			3	1679.84(7,337)	31.1
			4	1446.94(10,037)	23.8

trial spawns a catastrophic anomalous instance when $\delta = 0.3$. Therefore we can not adopt this trial nevertheless computation time improves when $\delta = 0.9$.

Second, we discard the formulae (6) and (7), and exploit the first clause (density-ordered) as same as the previous trial. As we can see in Table 11, no catastrophic anomalous instance could be spawned in the case where $\delta = 0.3$. In addition as to BBN, same performance as the usual in Table 7 can be shown. However this trial tends to spawn large BBN in the worst case, paying our attention to the case where $(\delta, n, |S|, m)$ is $(0.9, 60, 10, 2)$ in particular. Therefore this combination of the candidates is not preferable too.

Table 11: LP relaxation and density-ordered

δ	n	$\|S\|$	m	BBN(worst)	CPU (sec.)
0.3	60	10	2	198.32(705)	3.9
0.9	60	10	2	1008.12(11,423)	16.1
			3	976.68(5,309)	14.3
			4	882.84(2,583)	12.5
		20	2	1012.96(4,993)	30.4
			3	1444.44(8,623)	39.7
			4	1299.74(10,077)	34.3
		30	2	1239.20(17,319)	59.0
			3	1598.98(6,447)	71.4
			4	1515.90(7,233)	61.1

Third, we replace both the formulae (6) and (7) with each which takes N_0 into account respectively. The usual two initial multiplier vectors prescribed by the formulae (6) and (7) are constant on any node respectively, however, it would not reflect that several items might be already fixed to zero in the subproblem (9). Therefore we attempt to discard the value or the efficiency of items in N_0 and construct each initial multiplier vector by items in $N_1 \cup N_2$ respectively in this trial. Computation experiments are in Table 12. In this trial when $\delta = 0.9$ and $|S| = 10$, BBN in the worst case is well-bounded, which is a preferable behavior. In addition, computation time is better than the previous trial. This implies that the three types of initial multiplier vectors based on the greedy heuristics for IKP are a little expensive.

Table 12: LP relaxation and two new initial multiplier vectors

δ	n	$\|S\|$	m	BBN(worst)	CPU (sec.)
0.3	60	10	2	256.34(1,391)	4.3
0.9	60	10	2	780.08(4,315)	11.6
			3	1014.62(4,885)	13.2
			4	941.66(4,735)	11.5
		20	2	897.98(9,889)	20.4
			3	1244.20(11,191)	26.6
			4	1216.12(7,329)	23.6
		30	2	1159.56(23,493)	35.6
			3	1581.02(6,853)	44.0
			4	1595.80(8,643)	41.2

4.6 Uni-procedure

So far we have exploited two iterative procedures: one is *Bounds* for the top node, the other is *Iter_Proc* for the rest of all intermediate nodes. In this subsection we will apply the procedure *Bounds* to not only top node but also each intermediate node. In a word, only the *Bounds* is exploited in the implementation of our algorithm. In the latter half of this subsection we also discuss the experiment of exploiting the *Iter_Proc* only.

Repeating that we have adopted best-first search strategy and branching variable is determined as item $i \in N_2$ which gives maximal of $\sum_{s\in S} v_i^s$. The items in N_2 are sorted in a nonincreasing order of the $\sum_{s\in S} v_i^s$ beforehand. Then, branching variable is taken from the head of list N_2 with no condition. Also all nodes which are not expanded yet are stored in a list, and the nodes in the list are sorted in a nonincreasing order of the upper bound obtained by the subproblem of given MNK which is prescribed by the triplet of each node. Then, a node which should be expanded next is taken from the head of the list with no condition. In the initial state, before exploring a search-tree, there exists only one node in the list i.e. top node.

On each node, the *Bounds* is called six times with different types of initial multiplier vectors severally, and each solution giving LB or UB is reused respectively in each calling. Two initial multiplier vectors, the formulae (6) and (7), are always constant respectively, while the rest of four is only concerned with items in $N_1 \cup N_2$ respectively. The *Bounds* also includes the feature of dynamical step size.

Computation experiments are in Table 13. In the table each line expresses the average of 10 data instances, and shows the best result as to BBN among many trials. As we can see in Table 13 BBN is well-bounded, however, computation time is so terrible. This result has again revealed the importance of the "tradeoff" mentioned in [2].

In passing when $|S| = 30$ and $m = 2$ in Table 13, although Δ_1/Δ_2 is almost one accidentally (It is due to the lack of an anomalous instance), our algorithm costs ten times of BBN compared with the one in [5]. It would be owing to the difference of the sharpness of upper bound. As we have mentioned it at the end of Subsection 4.3, our lower bound is pretty good, however, upper bound is not so tight.

Table 13: Uni-procedure: *Bounds*

δ	n	$\|S\|$	m	BBN(worst)	CPU (sec.)	z_{XL}	z_{XU}	Δ_1	Δ_1/Δ_2
0.9	60	10	2	294.2(457)	168.3	2152.6	2186.0	0.0153	1.35
			3	433.4(651)	198.8	1846.2	1899.2	0.0279	1.50
			4	412.4(951)	189.2	1512.2	1574.5	0.0396	2.37
		20	2	241.0(445)	225.9	2093.0	2123.2	0.0142	1.35
			3	545.4(977)	506.3	1687.3	1749.1	0.0353	1.83
			4	410.2(801)	327.5	1459.2	1518.7	0.0392	1.47
		30	2	405.8(755)	577.4	1998.3	2036.8	0.0189	1.02
			3	643.4(1,207)	765.7	1671.3	1745.6	0.0426	2.55
			4	628.4(1,355)	713.7	1357.3	1424.0	0.0468	2.26

On the other hand applying the procedure *Iter_Proc* to all nodes can be also discussed. Then, we will here adopt the third trial in Subsubsection 4.5.2 as the processing which is applied to every node, including top node. From the point of view of computation time, it would be the best choice. In the trial, the procedure *Iter_Proc* is called three times with different types of initial multiplier vectors severally: $1/\bar{Z}_D(\mu_s)$; formulae (6) and (7) only concerned with items in $N_1 \cup N_2$ respectively. Also the procedure does not include the feature of dynamical step size.

Computation experiments are in Table 14. In the table each line expresses the average of 100 data instances. Compared with Table 7, the computation time indeed improves. Also the average of BBN is almost same compared with Table 7, however, it seems that it will tend to spawn a little large BBN in the worst case. It would be owing to the weakness of initial lower bound because of the lack of the procedure *Bounds* in the implementation. Except the defect, it could be concluded that this processing is sufficient to solve the MNK problem. In other words, nevertheless only three candidates for lower bound and four candidates for upper bound are exploited, this light processing is sufficient to solve the MNK problem in reasonable computation time when both n and $|S|$ are moderate.

While it does not mean that this study has finished, we will put a period to this study for the time being by adopting the computation result presented in Table 14 as the final result in this report.

Table 14: Uni-procedure: *Iter_Proc*

δ	n	$\|S\|$	m	BBN(worst)	CPU (sec.)	z_{XL}	z_{XU}	Δ_1	Δ_1/Δ_2
0.3	60	10	2	190.78(463)	3.0	2315.68	2334.50	0.0081	1.92
			3	276.12(1,267)	3.6	1888.64	1914.17	0.0133	5.11
			4	231.44(723)	3.0	1607.32	1638.67	0.0191	4.06
		20	2	256.62(923)	6.0	2279.71	2299.90	0.0088	2.51
			3	321.24(1,121)	7.3	1860.04	1887.90	0.0148	2.46
			4	295.82(855)	6.3	1587.44	1617.49	0.0186	4.53
		30	2	255.00(2,187)	8.1	2281.29	2302.36	0.0092	1.87
			3	320.94(1,123)	9.6	1823.97	1851.94	0.0151	2.55
			4	274.04(1,201)	7.9	1550.45	1581.96	0.0199	2.31
0.6	60	10	2	341.70(2,071)	4.9	2219.32	2251.32	0.0142	2.15
			3	494.18(3,507)	6.8	1803.75	1850.61	0.0253	2.38
			4	442.96(1,281)	5.7	1531.41	1582.46	0.0323	7.34
		20	2	406.60(3,131)	9.3	2149.33	2186.63	0.0171	1.87
			3	626.38(2,921)	13.2	1742.04	1793.76	0.0288	3.03
			4	428.56(2,037)	8.6	1499.29	1556.35	0.0367	3.52
		30	2	465.30(1,861)	13.8	2163.33	2200.26	0.0168	3.81
			3	696.72(4,495)	20.6	1748.49	1807.73	0.0328	2.32
			4	577.28(2,469)	15.8	1497.23	1558.01	0.0390	2.07
0.9	60	10	2	726.48(4,553)	9.1	2122.82	2179.98	0.0262	2.31
			3	882.92(5,889)	10.7	1743.43	1823.58	0.0440	2.36
			4	1099.22(7,743)	12.4	1514.28	1607.36	0.0579	3.46
		20	2	965.00(4,107)	20.0	2104.26	2179.34	0.0345	3.28
			3	1338.56(6,725)	26.0	1664.19	1764.02	0.0566	2.94
			4	1187.24(9,911)	21.4	1412.45	1510.45	0.0649	2.43
		30	2	1290.10(8,385)	36.9	2013.84	2091.58	0.0372	2.02
			3	1829.58(13,835)	48.0	1652.59	1754.82	0.0583	3.49
			4	1557.82(14,263)	38.9	1389.79	1506.62	0.0775	3.74

z_{XL}: lower bound by the maximal of the three candidates
z_{XU}: upper bound by the minimal of the four candidates
Δ_1 : $(z_{\mathrm{XU}} - z_{\mathrm{XL}})/z_{\mathrm{XU}}$
Δ_2 : $(z_U - z_L)/z_U$, as is in [5]

5 Conclusion

In this report we have studied the max-min 0–1 knapsack (MNK) problem. It has been revealed that the bounds proposed by Yu are no doubt stable especially against an instance of the MNK problem with large number of scenarios in contrast to ours. On the other hand we have demonstrated how our proposed algorithm is sufficient to solve the MNK problem in the case where both the number of items and scenarios are moderate, which is due to the lightness of the processing for the bounds. In other words in such a case, our proposed bounds are enough to solve the MNK problem in reasonable computation time nevertheless they are not so tight compared with those by Yu.

Finally, our proposed algorithm has merely followed the same course as the one by Yu, that is, constructing surrogate relaxation of given MNK and solving it to obtain the bounds which are exploited in branch-and-bound method. To explore the alternative is a challenging problem and has yet been left to us.

Acknowledgments

This study has been done during the period January to May in 1997 under the direction of Professor Masayuki Kimura as a sub-theme that is one of the requirements for the degree of doctor of philosophy in School of Information Science, JAIST Hokuriku.

References

[1] Dantzig, G.B., "Discrete variable extremum problems," *Opns. Res.* **5**, 266–277 (1957).

[2] Fisher, M.L., "The Lagrangian relaxation method for solving integer programming problems," *Management Science* **27**(1), 1–18 (1981).

[3] Glover, F., "Surrogate constraint duality in mathematical programming," *Opns. Res.* **23**(3), 434–451 (1975).

[4] Kohli, R. and Krishnamurti, R., "Joint performance of greedy heuristics for the integer knapsack problem," *Discr. Appl. Math.* **56**, 37–48 (1995).

[5] Yu, G., "On the max-min 0–1 knapsack problem with robust optimization applications," *Opns. Res.* **44**(2), 407–415 (1996).