Regular Paper

Heuristic and Exact Algorithms
for the Disjunctively Constrained Knapsack Problem

TAKEO YAMADA,[†] SEIJI KATAOKA[†] and KOHTARO WATANABE[†]

We are concerned with a variation of the knapsack problem (KP), where some items are incompatible with some others. As in ordinary KPs, each item is associated with profit and weight, we have a knapsack of a fixed capacity, and the problem is to determine the set of items to be packed into the knapsack. However, the knapsack is not allowed to include incompatible pairs. The knapsack problem with such an additional set of constraints is referred to as the disjunctively constrained knapsack problem (DCKP). We present a heuristic method as well as an implicit enumeration algorithm and an interval reduction method to solve DCKPs to optimality. Combining these, we are able to solve DCKPs with up to 1,000 items.

1. Introduction

We are concerned with a variation of the *knapsack problem* (KP) [9],[11], where some items are *incompatible* with some others. As in ordinary KPs, we have n items to be packed into a knapsack of capacity c. Let w_j and p_j denote the weight and profit of the jth item respectively. Without much loss of generality we assume the following.

A$_1$. Problem data w_j, p_j ($j = 1, \ldots, n$) and c are all positive integers.

A$_2$. $\sum_{j=1}^{n} w_j > c$ and $w_j \le c$ for all j, since otherwise the problem is trivial.

A$_3$. Items are arranged in non-increasing order of profit per weight, i.e.,

$$p_1/w_1 \ge p_2/w_2 \ge \ldots \ge p_n/w_n. \quad (1)$$

In addition, let $E \subseteq \{(i,j) \mid 1 \le i \ne j \le n\}$ denote the set of incompatible pairs, and $m := |E|$ is the number of such pairs. That is, if $(i,j) \in E$ items i and j are not allowed to be included in the knapsack simultaneously. This relation is assumed to be reflective, i.e., $(i,j) \in E \Leftrightarrow (j,i) \in E$. We call this *disjunctive* constraint, and the knapsack problem with these additional constraints is referred to as the *disjunctively constrained* knapsack problem (DCKP).

The problem is to fill the knapsack with items such that the capacity and disjunctive constraints are all satisfied and the *knapsack profit* is maximized, where by knapsack profit we mean the total profit of items in the knap-

sack. Let x_j be the decision variable such that $x_j = 1$ if item j is included in the knapsack, and $x_j = 0$ otherwise. Then, mathematically the problem is formulated as the following 0-1 programming problem [10],[13].

DCKP:

$$\text{maximize}\quad z(x) := \sum_{j=1}^{n} p_j x_j \quad (2)$$

$$\text{subject to}\quad \sum_{j=1}^{n} w_j x_j \le c, \quad (3)$$

$$x_i + x_j \le 1, \forall (i,j) \in E, \quad (4)$$

$$x_j \in \{0,1\}, j = 1, \ldots, n. \quad (5)$$

Here by X we denote the set of all feasible solutions, and $z(x)$ is the objective value for $x \in X$. Throughout the paper $x^\star$ denotes an optimal solution with the corresponding optimal objective value $z^\star := z(x^\star)$. **DCKP** is $\mathcal{NP}$-hard [5], since for $E = \emptyset$ it reduces to KP which is already $\mathcal{NP}$-hard.

DCKP may be solved using any free or commercial integer programming package such as LINDO, CPLEX and so on [4],[6], but in order to obtain optimal solution within reasonable CPU time instances must be of a limited size. To solve larger problems, we present a *heuristic* algorithm that gives a lower bound, derive upper bounds, and develop an algorithm to solve the problem to optimality. In particular, we introduce some pruning conditions, and based on these develop an *implicit enumeration* [7] algorithm. Combined with an *interval reduction* [2],[12] method, the developed algorithm is able to solve **DCKP**s with up to 1,000 items to optimality.

---

† Department of Computer Science, The National Defense Academy

2. Heuristic Algorithms

An arbitrary feasible solution $\underline{x} \in X$ naturally induces a lower bound $\underline{z} := z(\underline{x})$ to **DCKP**. In this section, we give a *greedy* [12] method that produces a good initial feasible solution quickly, and a *local search* [1] method that improves on the obtained solution.

The greedy algorithm consists of n steps, with the jth step corresponding to the decision for item j under the condition that $x_1, \ldots, x_{j-1}$ have been decided. That is, we put item j into the knapsack if the remaining capacity is large enough to accommodate it and no incompatible items have already been included. Thus, we have the following.

Algorithm GREEDY.

Step 1. Set $x := 0$ and $j := 0$;
Step 2. Set $j := j + 1$; If $j > n$ output $(\underline{x}^1, \underline{z}^1) := (x, z(x))$ and **stop**;
Step 3. If $\sum_{i=1}^{j-1} w_i x_i + w_j > c$ or $\exists i < j$, $x_i = 1, (i, j) \in E$, set $x_j := 0$; else set $x_j := 1$;
Step 4. Go to **Step 2**;

The output from this algorithm, $(\underline{x}^1, \underline{z}^1)$, is referred to as the *greedy solution*.

Next, for an arbitrary solution $x \in X$, we introduce its *2-opt neighborhood* $N(x)$ as the set of solutions obtained from x by changing the decision for at most two items. These are obtained from x by either one of the following operations, provided that the resulting solution is feasible. Here by $I(x)$ we denote the set of items included in the knapsack under soluion x, i.e., $I(x) := \{j | x_j = 1, 1 \le j \le n\}$.

(i)   Put an item into the knapsack, i.e., for $j \notin I(x)$ set $x_j := 1$.

(ii) Exchange a pair of items through the knapsack, i.e., for $i \in I(x)$ and $j \notin I(x)$, set $x_i = 0$ and $x_j = 1$.

The local search algorithm starts with the greedy solution, and repeats to move to a better neighbor until no such improvement is possible any further. The algorithm is as follows.

Algorithm LOCAL_SEARCH.

Input: $\underline{x}^1$ (greedy_solution);
Step 1. Set $x := \underline{x}^1$;
Step 2. If $\exists y \in N(x)$ such that $z(y) > z(x)$ go to **Step 3**; else output $(\underline{x}^2, \underline{z}^2) := (x, z(x))$ and **stop**;
Step 3. Set $x := y$ and go to **Step 2**;

The output from this algorithm, $(\underline{x}^2, \underline{z}^2)$, is referred to as the *2-opt solution*.

Example 1   Consider **DCKP** with $n = 10$, $c = 200$, $(p_j) = (32, 52, 64, 35, 96, 18, 66, 15, 13, 4)$, and $(w_j) = (3, 6, 9, 12, 38, 9, 77, 40, 96, 95)$. The incompatible pairs are $E = \{(1, 2), (1, 3), (1, 9), (3, 6), (3, 7), (4, 5), (4, 9), (5, 6), (7, 9), (8, 9), (8, 10)\}$. The greedy solution is $\underline{x}^1 = (1001011100)$ with $\underline{z}^1 = 166$, and in one iteration which exchanges items $1 \leftrightarrow 2$ it improves to the 2-opt solution $\underline{x}^2 = (0101011100)$ with $\underline{z}^2 = 186$.

3. Upper Bounds

An upper bound to **DCKP** can be found by replacing (5) with

$$0 \le x_j \le 1, \quad \forall j. \tag{6}$$

The resulting linear programming problem is easy to solve (although it is often time-consuming for large n and m), and the optimal objective value to this *continuous relaxation* is denoted as $\overline{z}_C$. Clearly $\overline{z}^1 := \lfloor \overline{z}_C \rfloor$ gives an upper bound to **DCKP**.

Alternatively, we may consider the following *Lagrangean relaxation* problem.

Lagrange:

$$\text{maximize} \quad L := \sum_{j=1}^{n} p_j x_j +$$
$$\sum_{(i,j) \in E} \lambda_{(i,j)} (1 - x_i - x_j) \tag{7}$$
$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \tag{8}$$
$$0 \le x_j \le 1, \forall j. \tag{9}$$

Here L is the Lagrangean for **DCKP**, and $\lambda_{(i,j)} \ge 0$ is the Lagrangean variable associated with (4). Rewrite L as

$$L = \sum_{j=1}^{n} (p_j - \sum_{e \in \partial_j} \lambda_e) x_j + \sum_{e \in E} \lambda_e \tag{10}$$

with

$$\partial_j := \{e \in E | j \text{ is incident to } e\}. \tag{11}$$

Then, for a fixed $\lambda := (\lambda_e)$ **Lagrange** is a *continuous* knapsack problem (possibly with some negative profits), which is easily solved. Let $x^\star(\lambda)$ be an optimal solution to **Lagrange** with the associated objective value $L^\star(\lambda) := L(x^\star(\lambda))$, and put $\overline{z}_L(\lambda) := \lfloor L^\star(\lambda) \rfloor$. In particular, $\overline{z}^2 := \overline{z}_L(0)$ corresponds to the case where disjunctive constraints (4) are all dropped.

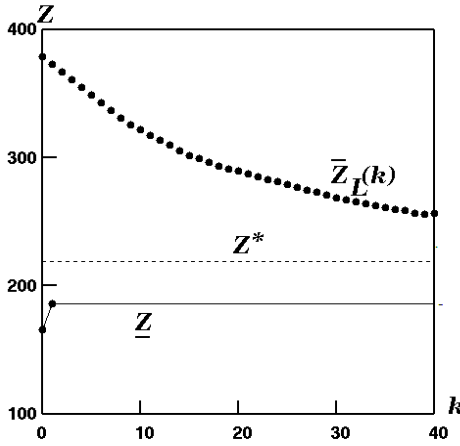From the duality theory for linear programming problems [3],[8], we have

Fig. 1 The behavior of Lagrangean relaxation.

$$\min\{L^{\star}(\lambda)|\lambda \geq 0\} = \overline{z}_C. \tag{12}$$

We note that $L^{\star}(\lambda)$ is a piece-wise linear convex function of λ, and for $e = (i, j)$

$$\partial L^{\star}(\lambda)/\partial \lambda_e = 1 - x_i^{\star}(\lambda) - x_j^{\star}(\lambda), \tag{13}$$

provided that $L^{\star}(\lambda)$ is differentiable at λ. Then, we tune up λ as follows: we start with $\lambda^0 := 0$, and at step k update it according to

$$\lambda^k = \lambda^{k-1} + \delta \cdot \partial L^{\star}(\lambda^{k-1})/\partial \lambda. \tag{14}$$

In our implementation we set $\delta = 1.0$, and the iteration stops if

$$L^{\star}(\lambda^k) \geq L^{\star}(\lambda^{k-1}) - 0.2 \tag{15}$$

is satisfied. Let the Lagrangean variable at this point be $\lambda^{\star}$. Then, we have an upper bound

$$\overline{z}_L := \overline{z}_L(\lambda^{\star}) \tag{16}$$

and the following is obvious.

Proposition 1

$$z^{\star} \leq \overline{z}^1 \leq \overline{z}_L \leq \overline{z}^2. \tag{17}$$

Thus $\overline{z}^2$ and $\overline{z}_L$ are weaker than $\overline{z}^1$ as upper bounds. However, the former are much easier to calculate, and especially $\overline{z}_L$ is often very close to $\overline{z}^1$.

Example 2 For the instance of Example 1, we have $\overline{z}^1 = 229$, $\overline{z}^2 = 378$, and $\overline{z}_L = 255$. **Figure 1** shows the behavior of the Lagrangean upper bound $\overline{z}_L(\lambda^k)$ through iteration. Also shown in the figure are the lower bounds $\underline{z}^1$, $\underline{z}^2$ and the optimal objective value $z^{\star}$.

4. Implicit Enumeration

This section presents an implicit enumeration algorithm to solve **DCKP** to optimality. This is further strengthened by an interval reduction method.

4.1 Complete Enumeration and Pruning

In the complete enumeration approach, we have two alternatives for each item:
(i)   put item j into the knapsack, or
(ii)   do not put it into the knapsack.

Thus, in total **DCKP** is solved in $O(2^n)$ steps. Since this is hopeless unless n is very small, we introduce some *pruning* methods. Let $(j, x_1, \ldots, x_j)$ be the node representing the state where $x_1, \ldots, x_j$ have been decided, and items $j + 1, \ldots, n$ are remaining to be included into the knapsack.

From node $(j, x_1, \ldots, x_j)$ two nodes $(j + 1, x_1, \ldots, x_j, 0)$ and $(j+1, x_1, \ldots, x_j, 1)$ are generated depending on the decision (i) or (ii) above for item $j + 1$. Then, starting with node $(0, -)$ we have a tree consisting of $\sum_{j=0}^n 2^j$ nodes. To prune some of the nodes (and their descendents) we introduce conditions for doing this.

Before stating the conditions, however, we prepare some terminology. At node $(j, x_1, \ldots, x_j)$, $F(j, x_1, \ldots, x_j)$ denotes the set of remaining items which is incompatible with some item already included in the knapsack, and thus can not be adopted. That is,

$$F(j, x_1, \ldots, x_j) := \{i|j + 1 \leq i \leq n, 1 \leq \\ \exists h \leq j, x_h = 1, (h, i) \in E\}. \tag{18}$$

Now, consider two nodes $(j, x_1, \ldots, x_j)$ and $(j, x_1', \ldots, x_j')$. We say the former is *dominated* by the latter if

$$\sum_{i=1}^j w_i x_i' \leq \sum_{i=1}^j w_i x_i, \tag{19}$$

$$\sum_{i=1}^j p_i x_i' \geq \sum_{i=1}^j p_i x_i, \tag{20}$$

and

$$F(j, x_1, \ldots, x_j) \supseteq F(j, x_1', \ldots, x_j') \tag{21}$$

hold.

Next, let $\overline{z}(j, x_1, \ldots, x_j)$ be an upper bound conditioned at $(j, x_1, \ldots, x_j)$, i.e., an upper bound to the problem with respect to items $j+1, \ldots, n$, under the condition that $x_1, \ldots, x_j$ have been determined. This can be obtained by slightly modifying either one of the upper bounds stated in Section 3.

Then, the following pruning conditions are straightforward.

Proposition 2 Node $(j, x_1, \ldots, x_j)$ can be terminated if either one of the following is satisfied.

C₁. $\sum_{i=1}^{j} w_i x_i > c$.

C₂. Item j is incompatible with some item already in the knapsack.

C₃. $(j, x_1, \ldots, x_j)$ is dominated by some other node.

C₄. $\overline{z}(j, x_1, \ldots, x_j) < \underline{z}$ for some lower bound $\underline{z}$ to **DCKP**.

These conditions eliminate weight-over nodes, incompatible nodes, dominated nodes, and unpromising nodes.

4.2 Implicit Enumeration Algorithm

With the pruning conditions **C₁**-**C₄**, the implicit enumeration algorithm can be constructed in the following manner. Here Ω_j represents the set of active nodes at the jth step, and a lower bound $\underline{z}$ is fed into the algorithm as an input.

Algorithm IMPLICIT_ENUM.

Input: A lower bound $\underline{z}$.

Step 1. Set $\Omega_0 := \{(0,\text{-})\}$ and $j := 0$;

Step 2. For all $(j, x_1, \ldots, x_j) \in \Omega_j$ do
if neither of **C₁**-**C₄** holds then
$\Omega_{j+1} := \Omega_{j+1} \cup \{(j+1, x_1, \ldots, x_j, 0), (j+1, x_1, \ldots, x_j, 1)\}$;

Step 3. If $\Omega_{j+1} = \emptyset$ **stop**.

**Step 4.** If $j < n - 1$ set $j := j + 1$ and go to **Step 2**.

Step 5. Find $(n, x_1, \ldots, x_n) \in \Omega_n$ such that $z := \sum_{i=1}^{n} p_i x_i$ is maximized;
Output $x^\star := (x_1, \ldots, x_n)$ and $z^\star := z(x^\star)$;

Remark 1 In implementing the above algorithm, we need to specify the conditional upper bound used in **Step 2** to see if **C₄** is satisfied. Since $\overline{z}^2$ is much faster than $\overline{z}_L$ to calculate, we evaluate $\overline{z}(j, x_1, \ldots, x_j)$ using $\overline{z}_L$ if 10 devides j, and $\overline{z}^2$ otherwise.

Example 3 For the instance of Example 1, **Fig. 2** depicts the tree of nodes generated by IMPLICIT_ENUM with an assumed lower bound $\underline{z} = 219$. Here ○ and ● at the terminal nodes mean that those nodes are terminated due to condition **C₄** and **C₂**, respectively. Conditions **C₁** and **C₃** are met at the nodes shown in the figure as C1 and C3. Thus in total 41 nodes are generated, and we obtain an optimal solution $x^\star = (0100101100)$ with $z^\star = 229$ at node 39 in the figure.

4.3 Interval Reduction Method

In IMPLICIT_ENUM we assumed that we feed a correct lower bound $\underline{z} \leq z^\star$ as an input. However, IMPLICIT_ENUM runs with an ar-
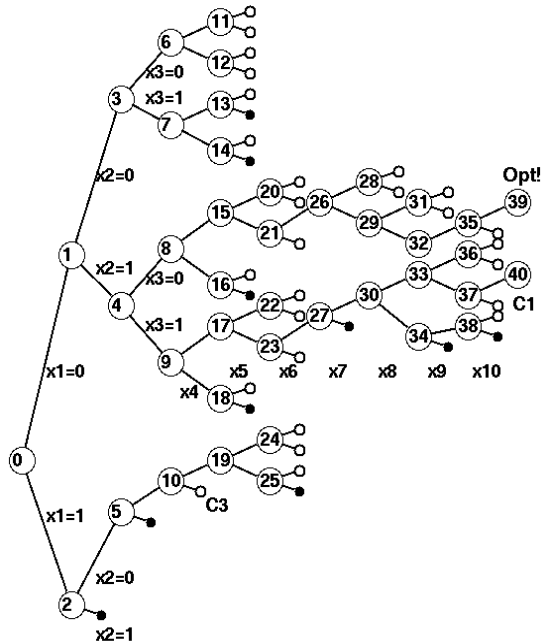
Fig. 2 The behavior of IMPLICIT_ENUM.

Table 1 The number of nodes generated by IMPLICIT_ENUM.

z	$P(z)$	Sol.	z	$P(z)$	Sol.
2879	43373	○	2915	1693	○
2880	41981	○	2916	1509	○
2890	23691	○	2917	1146	○
2900	11529	○	2918	901	×
2910	3748	○	2919	841	×
2911	3194	○	2920	794	×
2912	2711	○	2930	387	×
2913	2250	○	2940	362	×
2914	1864	○	2948	315	×

bitray value of z, irrespective to a lower bound or not. Such a z is referred to as a *trial value*. Then, if we feed a wrong trial value, i.e., $z > z^\star$, the algorithm eventually eliminates all the active nodes, meaning that $\Omega_j = \emptyset$ at some $j < n$, and ends up (in **Step 3**) without finding an optimal solution. Let $P(z)$ be the number of nodes generated by IMPLICIT_ENUM with z used as a trial value. The following is clear.

Proposition 3 IMPLICIT_ENUM correctly finds an optimal solution if and only if the trial value satisfies $z \leq z^\star$. Moreover, $P(z)$ is a non-increasing function of z.

Example 4 We consider the instance with $n = 200$, $c = 1000$, and p_j and w_j are uniformly random over $[1, 100]$. From 19900 pairs of items, 407 (approx. 2%) are randomly taken as incompatible pairs. The lower and upper bounds are $\underline{z}^2 = 2879$ and $\overline{z}_L = 2948$. With

Table 2 Result of numerical experiments for $c = 1000$.

η	n	m	$\underline{z}^2$	$\overline{z}_L$	$z^\star$	#rep	#nodes	CPU
	200	21.1	3488.5	3505.4	3502.6	1.5	2463.0	1.746
	400	80.6	4946.0	4958.7	4956.0	1.4	2173.4	1.349
0.001	600	180.6	6150.5	6157.4	6154.9	1.6	3083.8	11.375
	800	321.2	6997.0	7011.8	7007.6	1.4	9481.7	49.604
	1000	502.1	7788.1	7805.3	7802.8	1.5	14008.7	69.718
	200	39.9	3475.0	3493.4	3489.7	1.3	2426.4	1.774
	400	158.5	4894.8	4908.2	4905.3	1.9	4727.8	5.783
0.002	600	360.0	6043.1	6062.7	6060.5	1.5	11936.7	42.758
	800	640.8	6838.4	6864.8	6860.6	1.3	18269.7	161.675
	1000	1001.2	7709.8	7727.8	7722.5	1.7	10735.2	117.506
	100	26.8	2391.7	2423.4	2416.6	1.7	1624.3	0.384
	200	101.0	3361.9	3406.8	3396.1	2.1	6329.3	10.242
0.005	300	226.1	4067.6	4093.8	4088.0	1.6	5838.5	17.440
	400	394.7	4683.8	4731.7	4721.5	1.6	24543.3	152.175
	500	620.9	5210.7	5284.6	5279.1	1.1	57542.4	299.570
	100	50.8	2349.6	2384.3	2376.6	1.9	2001.2	0.731
	200	198.3	3246.8	3329.3	3309.7	1.8	7889.5	10.366
0.010	300	452.5	3892.7	3958.0	3946.0	1.2	13769.9	49.625
	400	802.1	4437.0	4525.4	4509.3	1.1	40372.3	249.040
	500	1249.1	4888.1	5022.8	4996.3	1.1	58221.1	356.404
	100	98.8	2249.1	2296.0	2279.6	1.8	1855.6	1.736
	200	387.1	3021.3	3157.2	3123.7	1.5	13590.6	27.353
0.020	300	888.0	3558.8	3706.7	3656.9	1.8	21550.5	125.606
	400	1600.4	4011.7	4232.5	4151.4	1.6	38128.1	381.033
	500	2494.6	4506.8	4718.7	4615.0	2.6	48840.3	722.668

$z = \underline{z}^2$, IMPLICIT_ENUM generates 43373 nodes and finds the optimal value $z^\star = 2917$ in 84.54 CPU seconds (on an HP9000 B132L workstation). If we happen to use, say $z = 2912$ as a trial value, we would have the same optimal solution with 2711 nodes generated in 2.7 seconds. **Table 1** shows $P(z)$ for some $z \in [\underline{z}^2, \overline{z}_L]$. Here $\bigcirc$ and $\times$ in the column of 'Sol'. show, respectively, that the optimal solution is, and is not, found for each z. These are divided into two disjoint groups depending on $z \le z^\star$ or $z > z^\star$.

The above observation leads us to the following *guess and try* approach. Here IM-PLICIT_ENUM runs with an etimated lower bound z (which is much larger than $\underline{z}$), and if it fails we try again with z lowered. Thus we have the following interval reduction method.

Algorithm INTERVAL_REDUCTION.

> **Input:** An upper bound $\overline{z}$ and a lower bound $\underline{z}$.
> **Step 1.** Set $z := \lfloor \alpha\overline{z} + (1-\alpha)\underline{z} \rfloor$;
> **Step 2.** Call IMPLICIT_ENUM with z as a trial value;
> **Step 3.** If an optimal solution is found in **Step 2**, output it and **stop**;
> **Step 4.** Set $\overline{z} := z$ and go to **Step 1**.

Here, $\alpha \in (0, 1)$ is a parameter of the algo-

rithm, and from some preliminary experiments we set this as $\alpha = 0.7$. As for the initial bounds, we may take $[\underline{z}, \overline{z}] := [\underline{z}^2, \overline{z}_L]$ discussed in Sections 2, 3. Then, at each iteration the interval $[\underline{z}, \overline{z}]$ shrinks by the factor of α, and an optimal solution is found in finite steps.

Example 5 For the instance of Example 4, we start with $\underline{z} = 2879$ and $\overline{z} = 2948$. With $\alpha = 0.7$, we first try $z = 2927$. Here IM-PLICIT_ENUM fails after generating 391 node. The upper bound is revised to $\overline{z} = 2927$, and we try again with $z = 2912$. This time we obtain the same optimal solution as in Example 4. Thus, in total we have solved the same instance with 3102 node and 2.95 CPU sec.

5. Numerical Experiments

We have implemented the algorithms of the previous sections in C language and conducted some numerical tests on an HP9000-B132L workstation to evaluate the performance of the developed algorithms. In the our experiments, the knapsack capacity is set to $c = 1000/2000$, and n ranges from 100 to 1000. The weights and profits are assumed random and independent over $[1, 100]$. Out of $n(n-1)/2$ pairs of items $m := \lfloor \eta \cdot n(n-1)/2 \rfloor$ pairs are randomly taken as incompatible pairs. Here η is the ratio of incompatible pairs, and we try the cases of $\eta = 0.001/0.002/0.005/0.01/0.02$.

Table 3   Result of numerical experiments for $c = 2000$.

η	n	m	$\underline{z}^2$	$\overline{z}_L$	$z^\star$	#rep	#nodes	CPU
	200	21.1	4965.7	4987.8	4985.1	1.7	5304.8	5.328
	400	80.6	7021.0	7035.5	7033.3	1.2	7173.5	16.314
0.001	600	180.6	8641.6	8659.7	8657.5	1.1	15611.7	60.409
	800	321.2	9855.9	9887.7	9881.3	1.2	43951.2	398.060
	1000	502.1	11074.1	11101.9	11097.8	1.2	45058.8	400.573
	200	39.9	4924.5	4946.6	4944.4	1.5	6688.3	9.439
	400	158.5	6893.0	6918.6	6916.5	1.0	19815.3	75.748
0.002	600	360.0	8456.3	8507.6	8498.5	1.1	57832.6	277.858
	800	640.8	9575.0	9634.7	9617.9	1.5	100674.1	1610.932
	1000	1001.5	10875.4	10915.6	10903.5	2.2	89741.6	1222.446
	100	26.8	3361.3	3418.1	3410.2	1.2	4246.2	2.825
	200	101.0	4703.2	4780.7	4770.3	1.6	26276.5	99.901
0.005	300	226.1	5692.1	5755.9	5745.2	1.5	35414.8	213.651
	400	394.7	6511.4	6600.9	6589.5	1.0	72295.3	511.414
	500	620.9	7230.1	7363.0	7340.5	1.2	183691.5	1609.440
	100	50.8	3268.1	3329.9	3314.4	1.9	4698.4	6.757
	200	198.3	4473.2	4611.1	4582.6	1.3	32506.8	155.524
0.010	300	452.5	5311.2	5461.9	5423.8	1.2	73862.0	539.006
	400	802.1	6023.1	6232.3	6180.4	1.1	94849.4	958.528
	500	1260.6	6556.6	6807.0	6718.5	1.7	179109.6	2148.943^a
	100	98.8	2975.0	3128.2	3066.4	2.0	8645.8	27.565
	200	387.1	4053.4	4303.0	4233.2	1.4	39697.2	240.164
0.020	300	888.0	4742.7	5058.7	4908.5	2.3	110531.5	981.217
	400	1611.0	5087.7	5471.0	5318.3	1.7	47927.3	678.260^b

a: Average of 8 successful runs. Two runs failed due to insufficient memory.
b: Average of 3 successful runs. Seven runs failed due to insufficient memory.

Table 2 and **Table 3** give the result of experiments of the implicit enumeration algorithm combined with the interval reduction method for $c = 1000$ and $c = 2000$, respectively. Here shown are the ratio (η) and the number of incompatible pairs (m), the lower and upper bounds $(\underline{z}^2$ and $\overline{z}_L)$ at the initial node of the tree, the optimal objective value $(z^\star)$, the number of iterations in interval reduction (#rep.), total number of nodes generated (#nodes), and CPU time in seconds. Each row is the average of 10 independent runs.

From these tables, we observe the following:
- We are able to solve problems with up to 1000 items within reasonable CPU time.
- The problem becomes harder to solve, with respect to both memory requirement and computation time, as n, m, and/or c increase.
- The number of iterations in interval reduction remains almost constant for various values of n, c and η.

Next, **Table 4** compares the CPU time (in seconds on an IBM RS/6000 44P Model 270) required of an integer programming solver employing branch and bound method (B&B) against ours (Impl_Enum). For B&B we used the FORTRAN code to solve general integer programming problems developed by Ibaraki

Table 4   Comparison against an integer programming solver.

c	α	n	B&B	Impl_Enum
		100	0.3	0.32
		200	102.3	0.87
	0.01	300	298.0	4.75
1000		400	-	33.47
		100	2.0	1.24
	0.02	200	166.7	2.95
		300	-	19.56
		100	1.1	3.57
		200	75.5	2.73
	0.01	300	165.6	75.78
2000		400	-	49.81
		100	1.2	20.03
	0.02	200	253.7	13.55
		300	-	58.42

and Fukushima[6]. Here dash (-) shows that the problem was unsolvable due to insufficient memory. Thus usually our method solves larger problems in smaller CPU time than the B&B approach.

6. Conclusion

We have formulated **DCKP** and developed both heuristic and exact algorithms. Especially, using the implicit enumeration algorithm combined with an interval reduction method we were able to solve problems with up to 1,000 items. To solve larger problems to opti-

mality, however, more sophisticated algorithms equipped with improved pruning methods are needed.

References

1) Aarts, E. and Lenstra, J.K. (Eds.): *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester, England (1997).

2) Baase, S.: *Computer Algorithms: Introduction to Design and Analysis*, 2nd ed., Addison-Wesley, Reading, Massachusetts (1993).

3) Chvátal, V.: *Linear Programming*, Freeman and Company, San Francisco, California (1983).

4) Fourer, R.: Software Survey: Linear Programming, *OR/MS Today*, Vol.26. pp.64–71 (1999).

5) Garey, M.R. and Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, CA (1979).

6) Ibaraki, T. and Fukushima, M.: *FORTRAN77 Optimization Programming* (in Japanese), Iwanami, Tokyo (1991).

7) Land, A.H. and Doig, A.G.: An Automatic Method for Solving Discrete Programming Problems, *Econometrica*, Vol.28, pp.497–550 (1960).

8) Luenberger, D.G.: *Linear and Nonlinear Programming*, 2nd ed., Addison-Wesley, Reading, Massachusetts (1984).

9) Martello, S. and Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Chichester, England (1990).

10) Nemhauser, G.L. and Wolsey, L.A.: *Integer and Combinatorial Optimization*, John Wiley & Sons, New York (1988).

11) Salkin, S.M.: The Knapsack Problem: A Survey, *Naval Research Logistic Quarterly*, Vol.27, pp.127–144 (1975).

12) Sedgewick, R.: *Algorithms in C*, 3rd ed., Addison-Wesley, Reading, Massachusetts (1998).

13) Wolsey, L.A.: Integer Programming, John Wiley & Sons, New York (1998).

Takeo Yamada received his B.S. degree from Kyoto Univ. in 1970, and M.S. and Ph.D. degrees from Stanford Univ. in 1980 and 1983 respectively. Since 1972 he has been with the National Defense Academy, where he is now a professor of computer science. His current research interests focus on mathematical programming, combinatorial optimization and computer algorithms to solve these problems. He has published extensively on these subjects in American and European professional journals, and received an Outstanding Paper Award from IEEE Control Systems Society in 1985. He is a member of ORSJ, JSIAM, JIMA, SICE and IEEE, as well as an International Advisory Board member of JOR.

Seiji Kataoka was born in 1961. He received his B.E., M.E. and Ph.D. degrees all from Waseda Univ. in 1985, 1987, and 1993 respectively. Currently he is an associate professor of computer science at the National Defense Academy. His research interest is in combinatorial optimization algorithms and their applications to actual problems. He has published in such journals as JORSJ, EJOR and this journal. He is a member of ORSJ, MPS and IPSJ.

Kohtaro Watanabe was born in 1965. He received his B.S. and M.S. degrees from Tokyo Institute of Technology in 1989 and 1991 respectively. He is now a research associate at the Department of Computer Science, the National Defense Academy. His current research interests are in inverse problems in partial differential equations and combinatorial optimization problems. He is a member of MSJ, JSIAM, IECIE and IPSJ.