JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS: Vol. 105, No. 3, pp. 659–676, JUNE 2000

Heuristic and Exact Algorithms for the Precedence-Constrained Knapsack Problem[1]

N. SAMPHAIBOON[2] AND T. YAMADA[3]

Abstract. The knapsack problem (KP) is generalized taking into account a precedence relation between items. Such a relation can be represented by means of a directed acyclic graph, where nodes correspond to items in a one-to-one way. As in ordinary KPs, each item is associated with profit and weight, the knapsack has a fixed capacity, and the problem is to determine the set of items to be included in the knapsack. However, each item can be adopted only when all of its predecessors have been included in the knapsack. The knapsack problem with such an additional set of constraints is referred to as the precedence-constrained knapsack problem (PCKP). We present some dynamic programming algorithms that can solve small PCKPs to optimality, as well as a preprocessing method to reduce the size of the problem. Combining these, we are able to solve PCKPs with up to 2000 items in less than a few minutes of CPU time.

Key Words. Combinatorial optimization, dynamic programming, knapsack problem, precedence constraints.

1. Introduction

Let $G_0 = (V_0, E_0)$ be a directed graph (Refs. 1–3) with vertex set

$$V_0 = \{v_1, v_2, \ldots, v_n\}$$

and edge set

$$E_0 = \{e_1, e_2, \ldots, e_m\} \subseteq V_0 \times V_0.$$

[1]This paper is dedicated to David Luenberger. The authors are grateful to Prof. H. Suzuki, University of Tsukuba, for informing them of his work as well as for helpful comments.

[2]Scientific Research Officer, Air Support Command, Royal Thai Air Force, Bangkok, Thailand.

[3]Professor, Department of Computer Science, National Defense Academy, Yokosuka, Kanagawa, Japan.

0022-3239/00/0600-0659$18.00/0 © 2000 Plenum Publishing Corporation

We assume that G_0 is acyclic; i.e., no directed cycles are included in G_0. Then, without loss of generality, we can assume that the vertices of G_0 are topologically sorted (Ref. 4), in the sense that

$$(v_i, v_j) \in E_0 \text{ implies } i < j.$$

Here, V_0 represents the set of items that can be included into a knapsack of capacity W. Associated with each $v_i \in V$ are its weight w_i and profit p_i. We assume that W is a positive integer, while w_i and p_i are nonnegative integers. The edge set E_0 stands for some precedence relations among the items. That is, $(v_i, v_j) \in E$ implies that item j can be adopted only when item i has been included in the knapsack. Such a problem may be encountered by a college student who wishes to earn as many credits as possible in four years. Usually, precedence relations exist among the subjects in the form of prerequisites.

This problem can be formulated mathematically as a 0–1 programming problem (Ref. 5). Let x_i be a variable such that

$$x_i = 1, \quad \text{if item } i \text{ is adopted,}$$
$$x_i = 0, \quad \text{otherwise.}$$

Then, we have the following precedence-constrained knapsack problem:

$$\text{(PCKP)} \quad \max \quad p(x) := \sum_{v_i \in V_0} p_i x_i,$$
$$\text{s.t.} \quad \sum_{v_i \in V_0} w_i x_i \leq W,$$
$$x_i \geq x_j, \qquad \forall (v_i, v_j) \in E_0,$$
$$x_i \in \{0, 1\}, \qquad \forall v_i \in V_0.$$

Here, without loss of generality, we assume that

$$w_1 < W \quad \text{and} \quad \sum_{v_i \in V_0} w_i > W,$$

since otherwise the problem is trivial. PCKP is $\mathcal{NP}$-hard (Ref. 6); because without the second group of constraints, it reduces to the knapsack problem (KP, Refs. 7–9), which is already $\mathcal{NP}$-hard.

Thus, KP can be regarded as a special subclass of PCKP with $E_0 = \varnothing$. Another important subclass of PCKP is the tree-knapsack problem (TKP, Refs. 10–12), where G_0 is a directed tree with root node v_1. Hirabayashi et al. (Ref. 13) formulated a tool-module design problem as a PCKP on bipartite graphs. Moriyama et al. (Ref. 14) generalized this into PCKP under the name of partially-ordered knapsack problem, and developed a branch-and-bound algorithm with some numerical experiments.

In this article, we present some dynamic programming algorithms that can solve small PCKPs to optimality, as well as a preprocessing method to reduce the size of the problem. Combining these, we are able to solve PCKPs with up to 2000 items in less than a few minutes of CPU time.

2. Recurrence Relations

In an acyclic graph $G_0 = (V_0, E_0)$, $v' \in V_0$ is a descendant of $v \in V_0$ if there exist an integer $k \geq 0$ and a sequence of vertices $v = v^0, v^1, \ldots, v^k = v'$ such that

$$(v^i, v^{i+1}) \in E_0, \qquad \text{for } i = 0, 1, \ldots, k-1.$$

This is denoted as $v \rightsquigarrow v'$, and v is an ancestor of v'. Note that v is a descendant (ancestor) of itself. Let the sets of all descendants and all ancestors of v be denoted as

$$D_v := \{v' \in V_0 \mid v \rightsquigarrow v'\},$$

$$A_v := \{v' \in V_0 \mid v' \rightsquigarrow v\}.$$

For a subset of vertices $V \subseteq V_0$, $G_0|_V$ denotes the subgraph of G_0 restricted to V with corresponding edge set

$$E := \{e \in E_0 \mid e \in V \times V\}.$$

If there is no confusion, $G_0|_V$ is simply denoted as $G = (V, E)$. By top(G), we mean the vertex with the smallest index in V, namely,

$$\operatorname{top}(G) = v_{\min\{i \mid v_i \in V\}}. \tag{1}$$

Subgraph G is said to be a singleton if $|V| = 1$, and is said to be empty if $V = \varnothing$.

Corresponding to an arbitrary subgraph $G = (V, E)$, we introduce a restriction of PCKP to G as the following problem:

$$
\begin{aligned}
(\text{PCKP}(G, w)) \quad \max \quad & p(x) := \sum_{v_i \in V} p_i x_i, \\
\text{s.t.} \quad & \sum_{v_i \in V} w_i x_i \leq w, \\
& x_i \geq x_j, \qquad \forall (v_i, v_j) \in E, \\
& x_i \in \{0, 1\}, \qquad \forall v_i \in V.
\end{aligned}
$$

This is also referred to as subproblem G for simplicity. Let the optimal objective value to this problem be $p^*(G, w)$. Then, clearly the optimal value

to PCKP is

$$p^* := p^*(G_0, W).$$

Note also that, if we fix

$$x_{\text{top}(G)} = 1$$

in PCKP(G, w), the objective value is $p_{\text{top}(G)}$ plus the optimal value from the remaining nodes $V \backslash \{\text{top}(G)\}$ with the reduced knapsack capacity $w - w_{\text{top}(G)}$. On the other hand, if we set

$$x_{\text{top}(G)} = 0,$$

all the descendants of top(G) are excluded, and we will have the subgraph with node set $V \backslash D_{\text{top}(G)}$.

Define the left and right children of G by

$$G_L := G|_{V \backslash \{\text{top}(G)\}}, \tag{2}$$

$$G_R := G|_{V \backslash D_{\text{top}(G)}}. \tag{3}$$

Then, the optimal values from these children are $p_{\text{top}(G)} + p^*(G_L, w - w_{\text{top}(G)})$ and $p^*(G_R, w)$, respectively. Thus, we have

$$p^*(G, w) = \max\{p_{\text{top}(G)} + p^*(G_L, w - w_{\text{top}(G)}), p^*(G_R, w)\}, \tag{4}$$

and the optimal decision for $x_{\text{top}(G)}$ is given by

$$x^*(G, w) := \begin{cases} 1, & \text{if } p_{\text{top}(G)} + p^*(G_L, w - w_{\text{top}(G)}) \geq p^*(G_R, w), \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

For the case of singleton, i.e., $V = \{v_i\}$, we have immediately

$$p^*(G, w) := \begin{cases} p_i, & \text{if } w \geq w_i, \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

with

$$x^*(G, w) := \begin{cases} 1, & \text{if } w \geq w_i, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

For the empty graph, we have

$$p^*(\emptyset, w) := 0. \tag{8}$$

Example 2.1. Consider the PCKP on the acyclic graph of Fig. 1 with weights and profits shown at each vertex as w_i/p_i. For subgraph

$G = (011111)$, we have

$$\text{top}(G) = v_2,$$

$$D_{v_2} = \{v_2, v_3, v_6\},$$

and the left and right children are

$$G_L = (001111), \qquad G_R = (000110),$$

respectively. Here, we employed a convention to identify subgraph $G = (V, E)$ with the index vector (g_i), defined by $g_i = 1 \Leftrightarrow v_i \in V$.

3. Dynamic Programming Algorithms

Consider again the problem of Fig. 1 with $W = 8$. Starting from G_0, let us create the left and right children of each subproblem as far as possible. For example, since $\text{top}(G_0) = v_1$, by fixing x_1 at 1 and 0 respectively, we obtain the left child $G_1 = (011111)$, and the right child which happens to be empty this case. Similarly, from G_1, we have $G_2 = (001111)$ and $G_8 = (000110)$ as its left and right children.

Figure 2 shows the tree obtained by completing the whole process, and Table 1 lists all the subproblems together with the vectors

$$p^*(G) := (p^*(G, 0), \ldots, p^*(G, W))$$

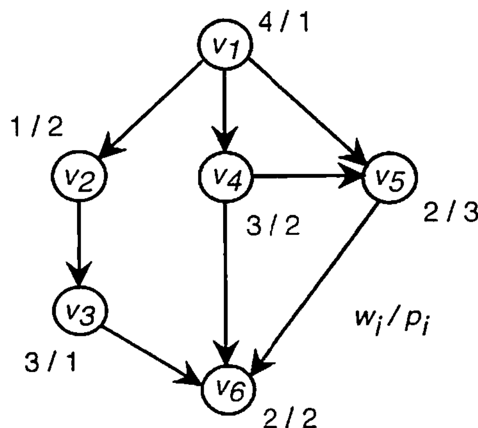


Fig. 1. PCKP for Example 2.1.

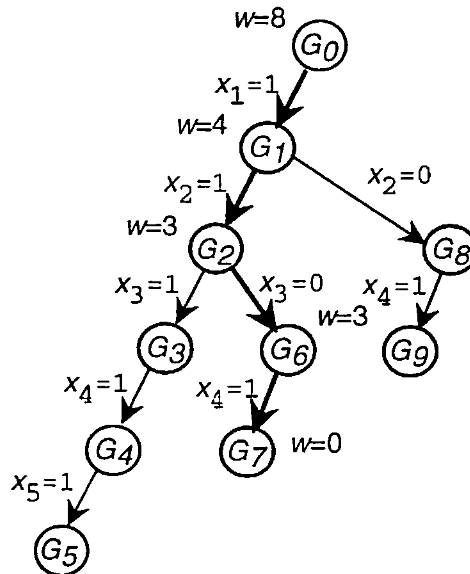


Fig. 2. Tree of subproblems for the PCKP of Fig. 1.

and

$$x^*(G) := (x^*(G,0), \ldots, x^*(G,W)).$$

Note that all terminal nodes of Fig. 2 are singletons for which $\{p^*(G), x^*(G)\}$ is easily found by (6)–(7). Then, these vectors can be propagated upward along the tree via (4)–(5), as shown in Table 1 where subproblems are listed as encountered in postorder traversal (Refs. 4 and 15) of Fig. 2.

Table 1. Vectors $p^*(G)$ and $x^*(G)$ subproblems in Fig. 2.

G	Index	$p^*(G)$	$x^*(G)$
G_5	000001	002222222	001111111
G_4	000011	003355555	001111111
G_3	000111	000225577	000111111
G_7	000010	003333333	001111111
G_6	000110	000225555	000111111
G_2	001111	000225556	000000001
G_9	000010	003333333	001111111
G_8	000110	000225555	000111111
G_1	011111	022245777	011110111
G_0	111111	000013335	000011111

The following algorithm generates automatically subproblems, construct the tree, and calculates the vectors $p^*(G)$ and $x^*(G)$ upward along the tree in a recursive way.

Algorithm 3.1. DP1.

Input. Subgraph G.
Output. Vectors $p^*(G)$ and $x^*(G)$.
Step 1. If G is a singleton, find $p^*(G)$ and $x^*(G)$ by (6)–(7) and return.
Step 2. Otherwise, do the operations below.

(i) Define subgraphs G_L and G_R by (2)–(3).
(ii) Find $p^*(G_L)$ and $x^*(G_L)$ by calling DP1 recursively with G_L.
(iii) Find $p^*(G_R)$ and $x^*(G_R)$ by calling DP1 recursively with G_R.

Step 3. Calculate $p^*(G)$ and $x^*(G)$ via (4)–(5).

Indeed, Table 1 is the output from this algorithm, and the optimal value is found to be $p^* = p^*(G_0, 8) = 5$.

Once the vectors $p^*(G)$ and $x^*(G)$ are obtained for all subproblems on the tree of Fig. 2, we find the optimal solution to PCKP by traversing the tree downward from G_0 in the following way.

Algorithm 3.2. Downward.

Input. Tree of subproblems with $p^*(G)$ and $x^*(G)$ for all subproblem G.
Output. Optimal solution x^* and optimal value p^*.
Comment. w denotes the remaining knapsack capacity at each node.
Step 1. Initialization. Set $w := W$, $G := G_0$, $x^* := 0$, $p^* := p(G_0, W)$.
Step 2. If $G = \varnothing$, output x^* and p^* and stop. Otherwise, if $x^*(G, w) = 1$, go to Step 3; else, go to Step 4.
Step 3. Set $i := \text{top}(G)$, $x_i^* := 1$, $w := w - w_i$, $G := G_L$ and go to Step 2.
Step 4. Set $G := G_R$, and go to Step 2.

Algorithms 3.1 and 3.2 constitute the forward and backward sweeps of the dynamic programming calculation (Refs. 16 and 17) along the tree of subproblems.

Example 3.1. For the PCKP of Example 2.1, the output from this algorithm is shown in Fig. 2 in bold arrows together with the value of w at each subproblem. Thus, we obtain the optimal solution $x^* = (110100)$ with $p^* = 5$.

One drawback with Algorithm DP1 is that it generates many identical subproblems repeatedly along the tree. For example, in Fig. 2, G_6 and G_8 are the same (see also Table 1), and all the descendents from the same subproblems are identical with each other. Thus, DP1 calculates the same vectors repeatedly along these subproblems. To avoid this, instead of calling DP1 recursively with G_L or G_R in Step 2 of DP1, we may check if the same subproblem has been generated already. In Fig. 2, before creating the right child of G_1, we see that the same problem has been generated as G_6. In such a case, we can refer to the previous results, as indicated by a link from G_1 to G_6 in Fig. 3, and thus reduce the number of subproblems generated.

To realize this savings in the number of subproblems, we need to keep the set of all previously generated subproblems. Let $\mathscr{G}$ denote this, which is initially $\mathscr{G} = \emptyset$. Then, the revised algorithm is as follows.

Algorithm 3.3. DP2.

Input. Subgraph G.
Output. Vectors $p^*(G)$ and $x^*(G)$.
Step 1. If G is a singleton, find $p^*(G)$ and $x^*(G)$ by (6)–(7) and return.
Step 2. Otherwise, do the operations below.
(i) Define subgraphs G_L and G_R by (2)–(3).

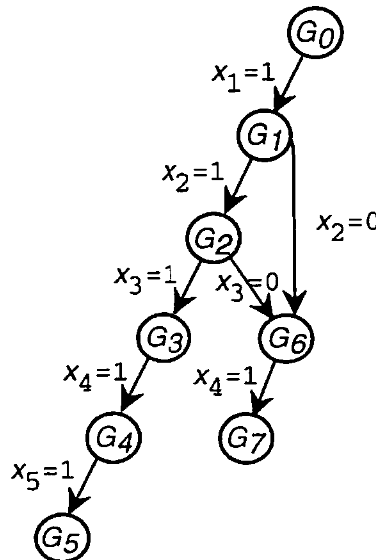


Fig. 3. Contraction of tree in Fig. 2.

(ii) If $G_L \in \mathscr{C}$, find $\{p^*(G_L), x^*(G_L)\}$ from $\mathscr{C}$ and go to (iii). Otherwise, call DP2 with G_L and obtain $\{p^*(G_L), x^*(G_L)\}$. Update $\mathscr{C}$ by inserting G_L together with the above obtained vectors.

(iii) If $G_R \in \mathscr{C}$, find $\{p^*(G_R), x^*(G_R)\}$ from $\mathscr{C}$ and go to Step 3. Otherwise, call DP2 with G_R to obtain $\{p^*(G_R), x^*(G_R)\}$. Update $\mathscr{C}$ by inserting G_R together with the above obtained vectors.

Step 3. Calculate $p^*(G)$ and $x^*(G)$ via (4)–(5).

In implementing Algorithm DP2, we need to specify the data structure for $\mathscr{C}$. This may be realized simply by using a linked list (Refs. 4 and 15). Alternatively, we may have $\mathscr{C}$ as a binary tree, where each subproblem is identified with the binary number representation of its index vector. We implemented Algorithm DP2 with $\mathscr{C}$ as a binary tree, since it takes $O(\log|\mathscr{C}|)$ time both to insert and retrieve a record. Usually, this outperforms other data structure for large $|\mathscr{C}|$.

4. Numerical Experiments: DP Algorithms

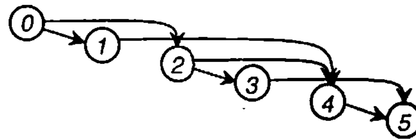
To evaluate the performance of the dynamic programming algorithms developed so far, we have conducted a series of numerical experiments on a variety of test problems. The instances tested are Random and Lattice as shown in Fig. 4. These are prepared as follows.

(a) Random. This consists of nodes $\{v_1, v_2, \ldots, v_n\}$: for all i, j satisfying $1 < i < j < n$, we create the edge (v_i, v_j) with probability $\lambda \in [0, 1]$. Next, all maximal vertices are connected from v_1 and all minimal ones are connected to v_n, so that v_1 [v_n] is the unique maximal [minimal] element in the graph.

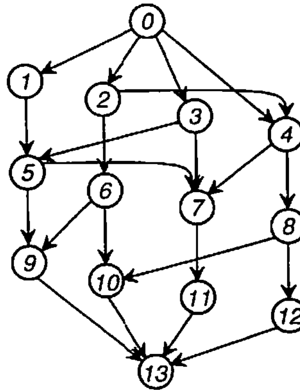
(b) Lattice. This consists of $s \times t$ lattice point nodes v_{it+j}, $i = 0, 1, \ldots, s-1$ and $j = 1, 2, \ldots, t$, and two additional nodes v_0 and v_{st+1}. First, we assume vertical edges $\{(v_i, v_{i+t}\}$ for $i = 1, 2, \ldots, (s-1)t$. Then, for $1 \leq i < j < i + t$, we create edge (v_i, v_j) with probability λ. Finally, v_0 is connected to $v_1, v_2, \ldots, v_t$, and $v_{st-t+1}, \ldots, v_{st}$ are linked to v_{st+1}, respectively.

The weights and profits are generated randomly and independently in [1, 100], and we have tested the cases of $\lambda = 0.2$ or 0.4 and $W = 10n$ or $25n$, for n ranging between $20 \leq n \leq 200$.

Table 2 summarizes the result of experiments for (a) Random instances, and Table 3 is for (b) Lattice instances. Here, we show the number of subproblems generated (#Sub) and the CPU time in seconds for the cases $W =$



(a)



(b)

Fig. 4. Test problems: (a) Random instance; (b) Lattice instance.

Table 2. Performance of DP algorithms for (a) Random instances.

		DP1			DP2		
λ	n	#Sub	CPU10	CPU25	#Sub	CPU10	CPU25
0.2	20	671.9	0.032	0.067	84.7	0.006	0.012
0.2	40	4124.4	0.481	0.948	335.4	0.078	0.128
0.2	60	9200.0	2.660	4.919	657.6	0.392	0.542
0.2	80	—	—	—	983.2	1.150	1.467
0.2	100	—	—	—	1243.4	2.856	3.341
0.2	150	—	—	—	2202.5	15.398	17.023
0.2	200	—	—	—	3009.0	49.631	—
0.4	20	89.0	0.005	0.007	43.8	0.002	0.005
0.4	40	226.6	0.048	0.072	109.3	0.033	0.049
0.4	60	364.1	0.234	0.307	173.4	0.161	0.197
0.4	80	574.2	0.771	0.927	247.6	0.497	0.565
0.4	100	612.5	1.587	1.804	288.3	1.118	1.220
0.4	150	987.0	8.340	8.860	437.7	5.579	5.829
0.4	200	1447.5	29.590	—	614.9	19.032	—

Table 3. Performance of DP algorithms for (b) Lattice instances.

				DP1			DP2		
λ	n	s	t	#Sub	CPU10	CPU25	#Sub	CPU10	CPU25
0.2	42	4	10	756.1	0.093	0.182	147.1	0.027	0.048
0.2	42	10	4	2410.6	0.297	0.581	329.4	0.075	0.123
0.2	62	6	10	2076.9	0.511	0.904	325.1	0.128	0.201
0.2	62	10	6	4955.0	1.104	2.065	482.2	0.213	0.323
0.2	82	4	20	1475.8	0.569	0.949	303.1	0.169	0.259
0.2	82	20	4	15801.4	7.682	12.628	948.9	0.912	1.217
0.2	102	5	20	3594.6	1.849	3.088	507.1	0.423	0.620
0.2	102	20	5	20473.3	15.152	24.390	1280.4	1.789	2.300
0.2	152	5	30	—	—	—	803.4	1.301	1.808
0.2	152	30	5	—	—	—	1988.6	7.302	8.581
0.2	202	8	25	—	—	—	1757.4	6.123	—
0.2	202	25	8	—	—	—	3029.1	18.133	—
0.4	42	4	10	166.1	0.025	0.044	89.4	0.017	0.030
0.4	42	10	4	200.8	0.039	0.064	107.1	0.028	0.043
0.4	62	6	10	309.4	0.097	0.154	154.5	0.068	0.099
0.4	62	10	6	363.6	0.128	0.195	165.5	0.088	0.122
0.4	82	4	20	374.7	0.167	0.261	191.5	0.115	0.170
0.4	82	20	4	518.6	0.465	0.600	228.3	0.297	0.361
0.4	102	5	20	584.9	0.384	0.574	265.3	0.252	0.349
0.4	102	20	5	715.7	0.933	1.168	307.7	0.603	0.717
0.4	152	5	30	896.5	1.175	1.656	399.0	0.754	0.984
0.4	152	30	5	962.8	3.703	4.225	451.3	2.569	2.833
0.4	202	8	25	1378.1	3.860	—	605.8	2.560	—
0.4	202	25	8	1358.9	8.169	—	606.4	5.434	—

$10n$ (CPU10) and $W = 25n$ (CPU25) with respect to algorithms DP1 and DP2. Each row is the average of 10 independent runs. The entries with dashes (–) show the cases which were unsolvable due to insufficient computer memory.

From these tables, the following conclusions emerge:

(i) DP2 is superior to DP1 in both CPU time and memory requirements.

(ii) For smaller λ, more subproblems are generated, and the CPU time is longer in both DP1 and DP2.

(iii) The number of subproblems generated is insensitive to W, and the CPU time increases only slightly as W increases.

(iv) DP2 is superior to DP1 for smaller λ. To see this, consider the ratio of the numbers of subproblems generated. (#Sub for DP2)/(#Sub for DP1). This is considerably smaller for $\lambda = 0.2$ than for $\lambda = 0.4$.

(v) In Lattice, problems with fewer rows ($s < t$) are easier to solve than problems with more rows ($s > t$).

With these DP algorithms, it is difficult to solve problems with more than a few hundred items, since these algorithms require memories to keep the $(W+1)$-dimensional vectors $p^*(G)$ and $x^*(G)$ for all the subproblems generated.

5. Heuristic and Reduction Algorithms

To solve larger problems, in this section, we propose a preprocessing method, consisting of a heuristic algorithm that finds quickly a lower bound to the optimal objective value and a reduction method to make the size of the problem smaller. The latter makes use of the lower bound and fixes considerably many variables at either 0 or 1. Thus, eliminating the fixed variables, we are left with problems of much smaller size that can be solved easily by the DP algorithms of the previous sections.

5.1. Greedy Algorithm. The following is a greedy algorithm that tries to adopt items, one by one, into the knapsack as far as possible.

Algorithm 5.1. Greedy.

Comment. p_G and w_G are the current knapsack value and weight.
Step 1. Set $x := 0$, $p_G := 0$, $w_G := 0$.
Step 2. Look for an item j such that:

(i) $x_j = 0$,
(ii) $x_i = 1$, $\forall (v_i, v_j) \in E_0$,
(iii) $w_G + w_j \leq W$.

If such an item j is found, go to Step 3. Otherwise, output $x_G := x$ and p_G; then, stop.

Step 3. Set $x_j := 1$, $p_G := p_G + p_j$, $w_G := w_G + w_j$, and go to Step 2.

The output (x_G, p_G) from this algorithm is referred to as the greedy solution.

Example 5.1. For the PCKP of Fig. 1, we have $x_G = (110100)$ with $p_G = 5$.

5.2. Reduction of the Problem. Let $x^* = (x_j^*)$ be an optimal solution to the PCKP. For each item i, we define

$$\bar{w}_i := \sum_{v_j \in A_{v_i}} w_j, \tag{9}$$

$$\tilde{p}_i := \sum_{v_j \in D_{v_i}} p_j. \tag{10}$$

Then, the following proposition is straightforward.

Proposition 5.1. (i) $\bar{w}_i > W$ implies $x_i^* = 0$; (ii) $\tilde{p}_i < p_G$ implies $x_i^* = 1$.

Proof.

(i) If $x_i^* = 1$, we have $x_j^* = 1$ for all $v_j \in A_{v_i}$. Then,

$$\sum_{v_j \in V} w_j x_j^* \geq \bar{w}_i > W,$$

which is a contradiction.

(ii) If $x_i^* = 0$, we have $x_j^* = 0$ for all $v_j \in D_{v_i}$. Then,

$$\sum_{v_j \in V} p_j x_j^* \leq \tilde{p}_i < p_G.$$

Therefore, x^* is not optimal, which is also a contradiction. □

This proposition serves as a pegging test (Ref. 18) to fix some of the variables at either 0 or 1, and thus reduce the size of the remaining problem.

Example 5.2. For the PCKP of Example 2.1, Table 4 gives $\bar{w}_i$ and $\tilde{p}_i$. From this and $p_G = 5$, we can fix $x_5^* = x_6^* = 0$ (superscript ♭ in Table 4) and

Table 4. Reduction of PCKP, Example 5.2.

Vertex	Ancestors	Descendants	$\bar{w}_i$	$\tilde{p}_i$
v_1	v_1	$v_1, v_2, v_3, v_4, v_5, v_6$	4	$0^\sharp$
v_2	v_1, v_2	v_2, v_3, v_6	5	6
v_3	v_1, v_2, v_3	v_3, v_6	8	8
v_4	v_1, v_4	v_4, v_5, v_6	7	$4^\sharp$
v_5	v_1, v_4, v_5	v_5, v_6	$9^\flat$	6
v_6	$v_1, v_2, v_3, v_4, v_5, v_6,$	v_6	$15^\flat$	9

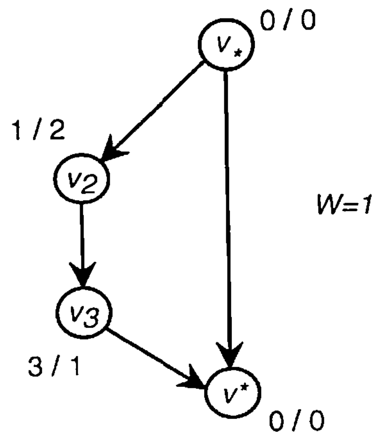


Fig. 5. Reduced PCKP.

$x_1^* = x_2^* = 1$ (superscript ♯ in Table 4); eliminating the fixed items, the problem is reduced to the PCKP of Fig. 5.

Example 5.3. As an example of larger problems, we solve a Random instance with $n = 200$, $\lambda = 0.4$, and $W = 2000$. By DP2, we obtain the optimal value $p^* = 1862$ in 19.8 s. On the other hand, we get a greedy solution with $p_G = 1676$, and with this, we fix 177 variables at 0 and 14 variables at 1, leaving a PCKP with only 9 variables. Solving this, we obtain again $p^* = 1862$, with the total CPU time of 0.30 s for greedy, reduction, and DP2 calculations.

Table 5 compares the CPU time with and without preprocessing. Each row is the average of 10 independent runs. From this, we conclude that preprocessing is quite effective in PCKP.

Table 5. CPU time (sec) of DP2 with and without preprocessing, Example 5.3.

			Preprocessed DP2		
n	p^*	DP2	Preprocessing	DP2	Total
60	673.9	0.16	0.029	0.000	0.029
100	1024.6	1.12	0.057	0.002	0.059
150	1461.6	5.56	0.148	0.004	0.152
200	1948.5	18.96	0.298	0.003	0.301

6. Numerical Experiments: Larger Problems

Table 6 summarizes the results of the numerical experiments for (a) Random instances with $\lambda = 0.2$ or 0.4, $W = 25n$, and n ranging from 500 to 2000. In addition to the independent case where w_j and p_j are uniformly and independently distributed in [1, 100], we have examined the correlated case, where p_j is related to w_j through $p_j = w_j + 100$. For each case, the percentage of variables fixed in reduction (Fixed), the size of the reduced problem (Size), and the CPU time in seconds are given. All entries are the average of 10 independent runs; dashes (–) indicate the cases that were unsolvable due to insufficient memory.

Table 7 gives similar results for (b) Lattice instances with $\lambda = 0.2$ or 0.4, $W = 25n$ and n ranging from 500 to 2000.

From these tables, we make the following observations:

(i) In most cases, preprocessing eliminates more than 95% of variables, and reduces problems of up to 2000 variables into PCKPs with a few dozens of items.

(ii) In solving PCKPs with preprocessing, most CPU time is consumed in the reduction part. Total CPU time is approximately proportional to λ.

(iii) In Lattice instances, problems with fewer rows ($s < t$) are easier to solve than problems with more rows ($s > t$), even with preprocessing.

(iv) Correlation between weights and profits is almost insensitive to the performance of the algorithms developed.

The last observation is in sharp contrast to many branch-and-bound based knapsack algorithms (Ref. 9), which are usually weak for strongly-correlated instances.

Table 6. Reduction of PCKP for (a) Random instances, $W = 25n$.

λ	n	Uncorrelated			Correlated		
		Fixed	Size	CPU	Fixed	Size	CPU
0.2	500	96.70	18.5	2.184	96.52	19.4	2.183
0.2	1000	97.94	22.6	16.863	98.14	20.6	16.856
0.2	1500	98.81	19.8	56.203	98.80	20.0	55.986
0.2	2000	99.24	17.1	132.402	99.06	20.9	132.334
0.4	500	99.32	5.4	4.281	98.96	7.2	4.299
0.4	1000	99.51	6.9	33.280	99.59	6.1	33.380
0.4	1500	99.71	6.4	111.648	99.71	6.4	112.352

Table 7. Reduction of PCKP for (b) Lattice instances, $W = 25n$.

λ	n	s	t	Uncorrelated			Correlated		
				Fixed	Size	CPU	Fixed	Size	CPU
0.2	502	50	10	96.53	19.4	0.526	95.84	22.9	0.591
0.2	502	10	50	96.40	20.1	0.250	97.35	15.3	0.233
0.2	1002	50	20	98.23	19.7	2.172	98.26	19.4	2.316
0.2	1002	20	50	98.46	17.4	1.294	98.15	20.5	1.306
0.2	1502	75	20	98.85	19.3	6.680	98.72	21.3	6.651
0.2	1502	20	75	98.81	19.8	2.815	98.87	19.0	2.771
0.2	2002	100	20	99.15	19.0	15.326	99.15	18.9	15.715
0.2	2002	20	100	99.20	18.0	5.673	99.11	19.8	5.667
0.4	502	50	10	99.18	6.1	0.861	98.80	8.0	0.854
0.4	502	10	50	99.04	6.8	0.282	99.26	5.7	0.292
0.4	1002	50	20	99.58	6.2	3.917	99.50	7.0	3.662
0.4	1002	20	50	99.57	6.3	1.837	99.44	7.6	1.947
0.4	1502	75	20	99.68	6.8	11.992	99.54	9.0	11.849
0.4	1502	20	75	99.74	5.9	4.040	99.71	6.3	4.346
0.4	2002	100	20	99.73	7.4	27.942	99.78	6.4	27.863
0.4	2002	20	100	99.80	6.0	8.344	99.74	7.2	8.859

7. Inverse PCKP

The inverse of PCKP is formulated as the following problem:

$$\begin{aligned}
\text{(IPCKP)} \quad \min \quad & w(x) := \sum_{v_i \in V_0} w_i x_i, \\
\text{s.t.} \quad & \sum_{v_i \in V_0} p_i x_i \geq P, \\
& x_i \geq x_j, \qquad \forall (v_i, v_j) \in E_0, \\
& x_i \in \{0, 1\}, \qquad \forall v_i \in V_0.
\end{aligned}$$

Similarly, subproblem IPCKP(G, p) is defined with respect to an arbitrary subgraph $G = (V, E)$ and a required level p of total profit. Let $w^\dagger(G, p)$ denote the optimal objective value to IPCKP(G, p), where $w^\dagger(G, p) := \infty$ if the problem is infeasible. Then, similar to (4), we obtain the recurrence relation

$$w^\dagger(G, p) = \min\{w_{\mathrm{top}(G)} + w^\dagger(G_L, p - p_{\mathrm{top}(G)}), w^\dagger(G_R, p)\}. \tag{11}$$

The optimal decision for $x_{\mathrm{top}(G)}$ is

$$x^\dagger(G, w) := \begin{cases} 1, & \text{if } w_{\mathrm{top}(G)} + w^\dagger(G_L, p - p_{\mathrm{top}(G)}) \leq w^\dagger(G_R, p), \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

Thus, again we obtain the same tree of subproblems as the one obtained in Section 3 (Fig. 2), and we can calculate the vectors

$$w^\dagger(G) := (w^\dagger(G, 0), \ldots, w^\dagger(G, P)),$$

$$x^\dagger(G) := (x^\dagger(G, 0), \ldots, x^\dagger(G, P))$$

by upward traversal of the tree. Next, we obtain an optimal solution $x^\dagger$ by tracing the tree downward. These constitute a DP algorithm to solve IPCKP, which may be further improved by the contraction method of Section 3.

The greedy method of Section 5 can be modified easily to find an approximate solution to IPCKP with a suboptimal objective value w_G. By definition, this gives an upper bound to the optimal value: i.e., we have $w_G \geq w^\dagger$. Then, corresponding to Proposition 5.1, we have the following proposition, which can be used as a pegging test for IPCKP.

Proposition 7.1. (i) $\bar{w}_i > w_G$ implies $x_i^\dagger = 0$; (ii) $\tilde{p}_i < P$ implies $x_i^\dagger = 1$.

8. Concluding Remarks

We have formulated PCKP and developed DP and reduction algorithms that can solve the problem with up to a few thousand items in a reasonable computing environment. However, to solve larger problems, further refinements of the algorithms are needed. Especially, recursion elimination from the DP algorithms may be worth trying, since programs with recursion tend to be inefficient in actual execution. Also, different approaches such as branch-and-bound methods should be explored.

References

1. Busacker, R. G., and Saarty, T. L., *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, NY, 1965.
2. Harary, F., *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969.
3. Ahuja, R. K., Magnanti, T. L., and Orlin, J. B., *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
4. Sedgewick, R., *Algorithms in C*, 3rd Edition, Addison-Wesley, Reading, Massachusetts, 1998.
5. Nemhauser, G. L., and Wolsey, L. A., *Integer and Combinatorial Optimization*, John Wiley and Sons, New York, NY, 1988.

6. Garey, M. R., and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, California, 1979.
7. Chvátal, V., *Linear Programming*, Freeman and Company, San Francisco, California, 1983.
8. Salkin, S. M., *The Knapsack Problem: A Survey*, Naval Research Logistic Quarterly, Vol. 27, pp. 127–144, 1975.
9. Martello, S., and Toth, P., *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley and Sons, New York, NY, 1990.
10. Johnson, D. S., and Niemi, K. A., *On Knapsacks, Partitions, and a New Dynamic Programming Technique for Trees*, Mathematics of Operations Research, Vol. 8, pp. 1–14, 1983.
11. Shaw, D. X., and Cho, G., *The Critical Item, Upper Bounds, and a Branch-and-Bound Algorithm for the Tree Knapsack Problem*, Networks, Vol. 28, pp. 205–216, 1998.
12. Cho, G., and Shaw, D. X., *A Depth-First Dynamic Programming Algorithm for the Tree Knapsack Problem*, INFORMS Journal on Computing, Vol. 9, pp. 431–438, 1997.
13. Hirabayashi, R., Suzuki, H., and Tuchiya, N., *Optimal Tool Module Design Problem for NC Machine Tools*, Journal of the Operations Research Society of Japan, Vol. 27, pp. 205–229, 1983.
14. Moriyama, H., Hada, T., and Suzuki, H., *A Partially-Ordered Knapsack Problem and Scheduling*, Proceedings of the Production Scheduling Symposium '96, Japan Industrial Management Association, pp. 79–84, 1996 (in Japanese).
15. Baase, S., *Computer Algorithms: Introduction to Design and Analysis*, 2nd Edition, Addison-Wesley, Reading, Massachusetts, 1993.
16. Bellman, R., *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957.
17. Larson, R. E., and Casti, J. L., *Principles of Dynamic Programming: Control and Systems Theory*, Vol. 7, Marcel Dekker, New York, NY, 1978.
18. Horowitz, E., and Sahni, S., *Computing Partitions with Applications to the Knapsack Problem*, Journal of the Association for Computing Machinery, Vol. 21, pp. 277–292, 1974.