INFORMATICA, 2016, Vol. 27, No. 2, 433–450

© 2016 *Vilnius University*
DOI: http://dx.doi.org/10.15388/Informatica.2016.93

Dynamic-Programming-Based Inequalities for the Unbounded Integer Knapsack Problem

Xueqi HE[1]*, Joseph C. HARTMAN[2], Panos M. PARDALOS[1]

[1]*Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611, USA*
[2]*Francis College of Engineering, University of Massachusetts Lowell, Lowell, MA 01854, USA*
e-mail: he0429@ufl.edu

Received: January 2015; accepted: April 2015

Abstract. We propose a new hybrid approach to solve the unbounded integer knapsack problem (UKP), where valid inequalities are generated based on intermediate solutions of an equivalent forward dynamic programming formulation. These inequalities help tighten the initial LP relaxation of the UKP, and therefore improve the overall computational efficiency. We also extended this approach to solve the multi-dimensional unbounded knapsack problem (d-UKP). Computational results demonstrate the effectiveness of our approach on both problems.

Key words: integer programming, dynamic programming, unbounded knapsack problem, valid inequalities.

1. Introduction

The unbounded knapsack problem (UKP) describes the following combinatorial optimization problem: given n types of items, items of type i have profit p_i, weight w_i, and unlimited supplies. The object is to determine the number of copies of each type of items that need to be placed in a knapsack, with capacity of c, to maximize total profit. Here, n, c, w_i and p_i, are positive integers. This can be formulated as:

$$\max \left\{ \sum_{i=1}^{n} p_i x_i : \sum_{i=1}^{n} w_i x_i \leqslant c,\ x_i \in Z^* \right\}. \tag{1}$$

The UKP, which was proved to be NP-hard by Lueker (1975), has been studied broadly in the integer programming literature. Floudas and Pardalos (2009) provided comprehensive overview of this problem. Gilmore and Gomory (1963) and Cabot (1970) proposed algorithms to solve the UKP based on branch and bound (B&B) method. By deriving upper bounds and defining a core problem, Martello and Toth (1990a) presented an algorithm for large UKP.

Various dynamic programming approaches have also been developed. Classic dynamic programming algorithms, which provide exact solutions, have been introduced by Dantzig

*Corresponding author.

(1957), Bellman (1957), and Gilmore and Gomory (1966). In order to reduce the search space and solution time, several important properties of the UKP have been studied and integrated into dynamic programming approaches. Gilmore and Gomory (1963, 1966) proposed the notions of dominance and periodicity. Based on partitioning in number theory, Yanasse and Soma (1987) provided a modified dynamic programming approach. Martello and Toth (1990b) introduced multiple dominance relations. Andonov *et al.* (2000) presented a new dynamic programming algorithm, EDUK, based on a new dominance relation termed threshold dominance.

By combining branch and bound approach and dynamic programming approach, several hybrid algorithms have been introduced to solve the UKP. Poirriez *et al.* (2009) provided an algorithm to incorporate information obtained from applying B&B approach to the core problem into dynamic programming. Martello and Toth (1984) presented a mixed approach to obtain the exact solution of the subset sum problem (SSP), which is a special type of the UKP.

The purpose of this paper is to illustrate how valid inequalities can be generated from solutions of intermediate stages of a dynamic programming algorithm to improve computational efficiency of solving the integer programming formulation of the UKP with traditional approaches. This approach was proposed by Hartman *et al.* (2010) for the capacitated lot-sizing problem. Unlike other hybrid algorithms, utilizing B&B approach to assist dynamic programming process, we incorporate inequalities derived from dynamic programming to the root node of branch and bound tree, since dynamic programming provides useful information to strengthen the integer programming formulation of the UKP. It is the first time, to the best of our knowledge, that such method has been applied to the UKP.

Moreover, this approach is adapted to solve the multi-dimensional unbounded knapsack problem (d-UKP). In reality, many applications of knapsack problems are consisted of more than a single constraint. The generalization of the UKP is the d-UKP, which could be described as follows:

$$\max\left\{\sum_{i=1}^{n} p_i x_i : \sum_{i=1}^{n} w_{ji} x_i \leqslant c_j,\ j \in \{1, \dots, d\},\ x_i \in Z^*\right\}. \tag{2}$$

Instead of a member of knapsack problems, the d-UKP was always considered as a special case of general integer programming problems, where all coefficients are positive and variables are non-negative. However, using the concept of knapsack, this particular type of problem could be handled effectively with our approach.

The rest of this paper is organized as follows. In Section 2, we review the concept of dominance relations for the UKP. Section 3 describes the classic dynamic programming method for solving the UKP and how we can revise it with dominance relations. Section 4 introduces our approach to obtain valid inequalities from the dynamic programming method. Section 5 discusses computational experiments design and shows test results. In Section 6, our approach is extended to solve the d-UKP. Finally, we conclude in Section 7.

2. Dominance Relations

Dominance relation is an important structural property for the UKP. Several researchers have contributed to this topic, such as Martello and Toth (1990a) and Andonov *et al.* (2000). We summarize known dominance relations as following:

1. Item type j **simply dominates** type i, if $w_j \leqslant w_i$ and $p_j \geqslant p_i$.
2. Item type j **multiply dominates** type i, if $\beta_j w_j \leqslant w_i$ and $\beta_j p_j \geqslant p_i$, where $\beta_j \in Z^+$.
3. A set of item types J **collectively dominates** type i, if $\sum_{j \in J} \beta_j w_j \leqslant w_i$ and $\sum_{j \in J} \beta_j p_j \geqslant p_i$, where $\beta_j \in Z^+$. Simple dominance and multiple dominance are special cases of collective dominance.
4. A set of item types J **threshold dominates** type i, if $\sum_{j \in J} \beta_j w_j \leqslant \alpha w_i$ and $\sum_{j \in J} \beta_j p_j \geqslant \alpha p_i$, where $\beta_j \in Z^+$ and $\alpha \in Z^+$.

Dominance relations help to simplify the problem. They set an upper bound for the number of copies of an item type in the optimal solution. If item type i is collectively dominated, it will generate less profit and consume more capacity compared with a combination of items in set J. Therefore, it will not appear in any optimal solution for any capacity, and we can discard it. If item type i is threshold dominated, for the similar reason, the maximum number of copies of type i in optimal solution is $\alpha - 1$. Detecting dominance relations could be considered as a step of pre-fixing variables and pre-processing to reduce the size of problem, which will decrease the complexity of following procedures.

3. Dynamic Programming Approach

3.1. *General Dynamic Programming*

Several dynamic programming approaches have been developed for the UKP. Garfinkel and Nemhauser (1972) introduced several fundamental dynamic programming approaches. In the consideration of how valid inequalities could be generated, following forward dynamic programming recursion is applied to our proposed method: let $f_i(y)$ be the maximum total profit that can be achieved in (1) with knapsack capacity y using only the first i type items, where $y = 0, 1, \dots, c$ and $i = 1, 2, \dots, n$. With initial conditions $f_i(0) = 0$, $\forall i = 1, 2, \dots, n$ and $f_0(y) = 0$, $\forall y = 0, 1, \dots, c$, the dynamic programming recursion is given by:

$$f_i(y) = \begin{cases} f_{i-1}(y), & \text{for } y < w_i. \\ \max\{f_{i-1}(y),\ p_i + f_i(y - w_i)\}, & \text{for } y \geqslant w_i. \end{cases} \tag{3}$$

The UKP could be solved by filling a $c \times n$ matrix with recursive equations given above. Each column i, $i \in \{1, 2, \dots, n\}$ represents a stage. While cell (y, i) represents a state and records the best profit that can be achieved by the first i type objects when y units of capacity have been used, or $f_i(y)$. The value in cell (c, n) shows the objective

value of the optimal solution. The complexity of this dynamic programming algorithm is $O(cn)$. When solving large-size problems, this algorithm could be time and memory space consuming.

3.2. *Modified Dynamic Programming*

Now, we will illustrate how dominance relations could be involved in dynamic programming and benefit the process. The detailed procedure is listed below:

Step 1. Identify the maximum weight among all item types that are in the dynamic programming process.

Step 2. Apply the dynamic programming algorithm, which is mentioned above, for all item types until capacity y reaches the maximum weight.

Step 3. Examine state $f_i(w_i)$ for each stage i. If $f_i(w_i) > p_i$, then item type i is collectively dominated. We add the upper bound constraint of $x_i \leqslant 0$ to the integer programming formulation and remove type i (stage i) from dynamic programming process.

Step 4. Complete the dynamic programming process on the remaining stages to the knapsack capacity c.

4. Valid Inequalities Derivation

Although, from a practical point of view, adjusted dynamic programming process runs faster than the original one, they have the same complexity $O(cn)$. When the capacity of a knapsack or the amount of item types is large, it may be cumbersome to execute the entire adjusted dynamic programming. With partial dynamic programming completed, information on intermediate solutions can be translated into effective inequalities for the integer programming formulation (1) of the UKP. To assure meaningful solutions, we sort items in descending order of profitability, i.e.:

$$\frac{p_1}{w_1} \geqslant \frac{p_2}{w_2} \geqslant \cdots \geqslant \frac{p_n}{w_n}.$$

It should be clear that an item with higher profitability is more likely to benefit the total profit of a knapsack when the capacity is fixed. And according to periodicity, another well-known property of the UKP, the most profitable item plays a different role with other type items, i.e. it is the only item that continuously contributes to the optimal solution when the capacity is above a threshold value. Therefore, we would like to place the most profitable item as the first stage.

Lemma 1 defines a valid inequality for the UKP based on the dynamic programming solutions.

Lemma 1. *For any $i = 1, 2, \ldots, n$, the following inequality is valid for all feasible solutions of* (1):

$$p_1x_1 + p_2x_2 + \cdots + p_ix_i \leqslant f_i(c). \tag{4}$$

Proof. By the definition of $f_i(y)$, we have

$$p_1x_1 + p_2x_2 + \cdots + p_ix_i \leqslant f_i(w_1x_1 + w_2x_2 + \cdots + w_ix_i).$$

Because $x_1, x_2, \ldots, x_i$ constitute a feasible solution, they should satisfy the problem constraint:

$$w_1x_1 + w_2x_2 + \cdots + w_ix_i \leqslant c.$$

Therefore, together with the fact that $f_i(y)$ is a non-decreasing function of capacity y, we conclude that

$$p_1x_1 + p_2x_2 + \cdots + p_ix_i \leqslant f_i(w_1x_1 + w_2x_2 + \cdots + w_ix_i) \leqslant f_i(c).$$

Thus, (4) is valid to (1). □

This inequality defines an upper bound on the profit that the first i types of items would contribute to the knapsack in all feasible cases.

Secondly, we introduce another valid inequality which defines an upper bound for the total profit of the first i types of items as a function of their total weight. Define $W_i = \sum_{k=1}^{i} w_kx_k$ and $P_i = \sum_{k=1}^{i} p_kx_k$. There exists a point (W_i, P_i) corresponding to each feasible solution and all of these points are on or below the step function $f_i(y)$ (refer to Figs. 1–3), because $f_i(y)$ can also be interpreted as the highest profit when the total weight of first i types of items is no more than y. Note that $f_i(y)$ is not necessarily a concave function of y. However, we could define the upper concave envelope of $f_i(y)$ by using J_i inequalities. Let a_{ij} and b_{ij} represent the slope and intercept of jth inequalities respectively, and these inequalities are in the following form:

$$f_i(y) \leqslant a_{ij}y + b_{ij}. \tag{5}$$

Since all feasible solutions (W_i, P_i) satisfy inequality (5), the upper bound on the total profit as a function of total weight can be defined as follows:

$$P_i \leqslant a_{ij}W_i + b_{ij}, \tag{6}$$

or

$$\sum_{k=1}^{i}(p_k - a_{ij}w_k)x_k \leqslant b_{ij}. \tag{7}$$

Since the knapsack has a capacity of c, y could be any value between 0 and c. To find the upper concave envelope of $f_i(y)$, we need to examine c points. This process could be simplified with the following two lemmas.

Lemma 2. *If the capacity of a knapsack is large enough such that $c \geqslant w_1$, then the first inequality which defines the concave envelope of $f_i(y)$ has slope $a_{i1} = \frac{p_1}{w_1}$ and intercept*

$b_{i1} = 0$, *or*:

$$f_i(y) \leqslant \frac{p_1}{w_1} y. \tag{8}$$

Proof. Recall that we have ordered the items by decreasing profitability. Because $f_i(y)$ records the best profit under capacity y and the highest profit per unit is given by $\frac{p_1}{w_1}$, $f_i(y)$, as well as the total profit for any feasible solution, should be less or equal to $\frac{p_1}{w_1} y$. Therefore, inequality (8) holds. Also we could confirm that (8) is tight and defines the first inequality of the concave envelope of $f_i(y)$, since it passes through points (0, 0) and (w_1, p_1) which are on the function $f_i(y)$. □

Lemma 3. *If there is no item that has the same profitability as item 1, then the second inequality segment begins from point* $(\lfloor \frac{c}{w_1} \rfloor w_1, \lfloor \frac{c}{w_1} \rfloor p_1)$.

Proof. The proof is straightforward. Because all points (kw_1, kp_1), $k = 0, \ldots, \lfloor \frac{c}{w_1} \rfloor$ are on the first inequality segment $f_i(y) = \frac{p_1}{w_1} y$ and all other items have smaller profitabilities, the second segment will begin at the last point on (8), i.e. the point $(\lfloor \frac{c}{w_1} \rfloor w_1, \lfloor \frac{c}{w_1} \rfloor p_1)$. □

Thus, we could obtain the first inequality directly, and to derive the remaining inequalities, only points between $\lfloor \frac{c}{w_1} \rfloor w_1$ and c, which are less than w_1 points, need to be investigated.

The following example illustrates how we can create the valid inequalities that we introduced above.

Example 1. Consider the UKP with three items with profits $p_1 = 15$, $p_2 = 9$, $p_3 = 4$, respectively, and weights $w_1 = 6$, $w_2 = 4$, $w_3 = 3$, respectively, and the knapsack has a capacity of 15. This problem can be formulated as:

$$\max \{15x_1 + 9x_2 + 4x_3 : 6x_1 + 4x_2 + 3x_3 \leqslant 15, x_1, x_2, x_3 \in Z^*\}.$$

The optimal solution is $x = (2, 0, 1)$ with a total profit of 34. Note that no item type is collectively dominated in this problem.

Solving the first stage, $i = 1$, with dynamic programming, we find $f_1(15) = 30$, such that the maximum profit can be achieved if the first item is 30. The valid inequality corresponding to (4) is:

$$15x_1 \leqslant 30.$$

Figure 1, which depicts $f_1(y)$ as a function of y, illustrates that $f_1(y)$ is non-linear and is an upper bound for all feasible solutions when considering only the first item type. We can derive the concave envelope inequalities for stage 1 according to (6). Connecting total weight 0 through 12 defines:

$$15x_1 \leqslant \frac{5}{2}(6x_1),$$

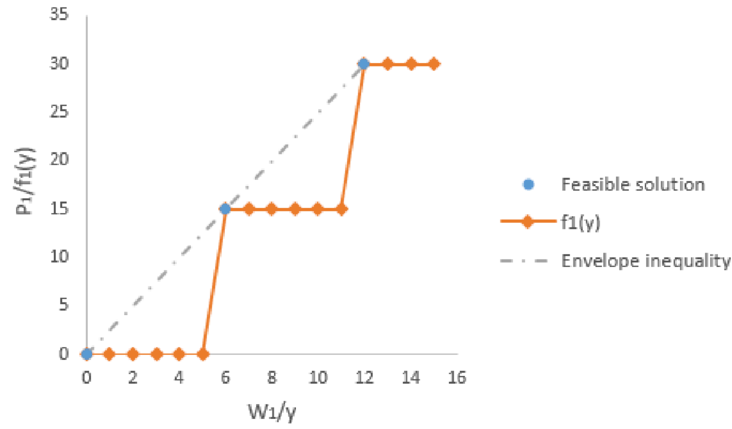

Fig. 1. Graph of $f_1(y)$ values and envelope inequalities.

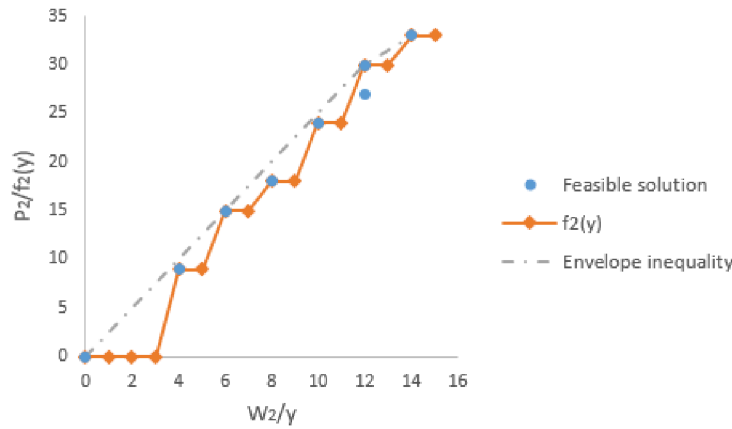

Fig. 2. Graph of $f_2(y)$ values and envelope inequalities.

while connecting total weight 12 through 15 defines:

$$15x_1 \leqslant 30.$$

For the first stage, the valid inequality obtained according to (6) is either redundant or a duplication of the valid inequality obtained according to (4). We generate it here to achieve the completeness of example. However, when solving problems, valid inequalities corresponding to (6) will not be derived for the first stage.

After solving stage 2, $i = 2$, we find $f_2(15) = 33$, and the inequality based on (4) is:

$$15x_1 + 9x_2 \leqslant 33.$$

Figure 2 includes information of the first two items and the concave envelope is defined with inequalities:

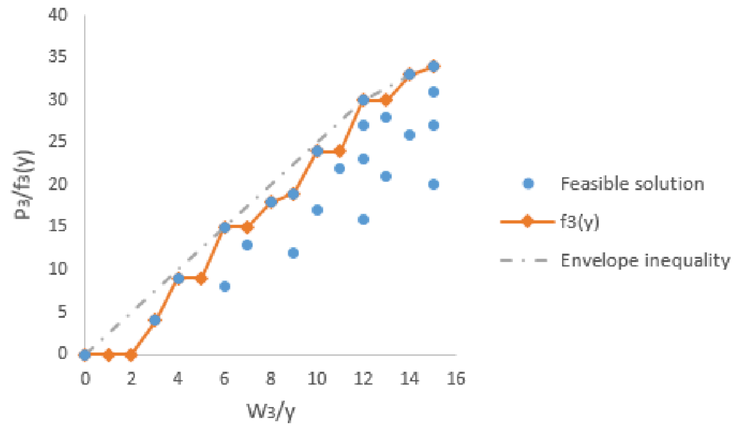

Fig. 3. Graph of $f_3(y)$ values and envelope inequalities.

$$
\begin{aligned}
15x_1 + 9x_2 &\leqslant \frac{5}{2}(6x_1 + 4x_2),\\
15x_1 + 9x_2 &\leqslant \frac{3}{2}(6x_1 + 4x_2) + 12,\\
15x_1 + 9x_2 &\leqslant 33.
\end{aligned}
$$

After stage 3, $i = 3$, the problem has been solved to optimality with $f_3(15) = 34$. The inequality described by Eq. (4) is:

$$15x_1 + 9x_2 + 4x_3 \leqslant 34,$$

and the concave envelope inequalities captured by Fig. 3 are:

$$
\begin{aligned}
15x_1 + 9x_2 + 4x_3 &\leqslant \frac{5}{2}(6x_1 + 4x_2 + 3x_3),\\
15x_1 + 9x_2 + 4x_3 &\leqslant \frac{3}{2}(6x_1 + 4x_2 + 3x_3) + 12,\\
15x_1 + 9x_2 + 4x_3 &\leqslant 6x_1 + 4x_2 + 3x_3 + 19.
\end{aligned}
$$

5. Computational Results

In this section, several data sets which involve different weight-profit correlation were generated to demonstrate the effectiveness of our proposed method.

5.1. *Instance Generation*

Random Cases. In accordance with the relationship of items' weights and profits, three types of random data sets were generated and tested with our presented approach: un-

correlated, weakly correlated and strongly correlated. This testing scheme was suggested by Martello and Toth (1990b). In all three cases, weights are randomly selected among the interval $[w_{\min}, w_{\max}]$. In the uncorrelated case, profits are randomly picked from $[p_{\min}, p_{\max}]$, so that the data sets of weights and profits are independent of each other. For the weakly correlated case, profit p_i is obtained randomly in the range of $[aw_i - b, aw_i + b]$, where a and b are two predetermined numbers. Finally, in the strongly correlated case, profit p_i is fixed as $aw_i + b$, which is linearly dependent on w_i.

In our experiments, both $n = 100$ and $n = 500$ have been examined. The ranges of items' weights $[w_{\min}, w_{\max}]$ varied based on the total number of items to allow more weight choices when n is big. Specifically for all random cases, we set parameters as follows: $w_{\min} = 30$, $w_{\max} = 200$, and $c = 0.2 \times \sum_{i=1}^{n} w_i$ when $n = 100$, while $w_{\min} = 60$, $w_{\max} = 800$, and $c = 0.1 \times \sum_{i=1}^{n} w_i$ when $n = 500$. For uncorrelated cases, $p_{\min} = 50$ and $p_{\max} = 250$ when $n = 100$, while $p_{\min} = 80$ and $p_{\max} = 850$ when $n = 500$. For weakly correlated cases and strongly correlated cases, we have $a = 1$ and $b = 20$ when $n = 100$, while $a = 1$ and $b = 40$ when $n = 500$.

Realistic Cases. Although the above random cases (uncorrelated, weakly correlated cases) create possible data sets for the UKP, they do not frequently happen in real world problems, since it is not common for an item with a heavier weight to be less valuable. Sinha and Zoltners (1979) first introduced a realistic scenario where a heavier item has more profit. To begin this data generation process, n weights and n profits are selected randomly from their own feasible intervals, $[w_{\min}, w_{\max}]$ and $[p_{\min}, p_{\max}]$. Then sort them in the descend order respectively and match sorted weight w_i and p_i up to become properties of one item type. At the end, the sequence of item types is reorganized according to the original order of items' weights. Hence, if $w_i \leqslant w_j$, then $p_i \leqslant p_j$ and weights w_i are in random order.

We run tests for realistic cases with the same parameter setting as uncorrelated random cases.

Hard Cases. Andonov *et al.* (2000) described three hard cases for the UKP concerning dominance. To construct these data sets, choose weights randomly in the range of $[w_{\min}, w_{\max}]$ firstly, where $w_{\min}$ and $w_{\max}$ are positive integers and satisfy $w_{\min} < w_{\max} < 2w_{\min}$. Then sort weights in the increasing order, and apply one of the following three equations to determine the profits of items. Finally, rearrange items according to the original order of weights.

$$p_i = w_i, \tag{9}$$

$$p_i = \max\left\{1 + p_{i-1}, \left\lfloor \frac{w_i p_{i-1}}{w_{i-1}} \right\rfloor\right\}, \quad p_1 \text{ is randomly chosen,} \tag{10}$$

$$p_i = \left\lfloor \frac{w_i p_{i-1}}{w_{i-1}} \right\rfloor + i - 1, \quad p_1 \text{ is randomly chosen.} \tag{11}$$

Hard cases satisfying (9)–(11) were tested respectively. When $n = 100$, we set $w_{\min} = 200$, $w_{\max} = 399$ and $c = 0.2 \times \sum_{i=1}^{n} w_i$, and when $n = 500$, we set $w_{\min} = 650$, $w_{\max} =$

Table 1
Time consumption for various testing cases (in ms).

Model	Uncorrelated		Weakly correlated		Strongly correlated		Realistic		Hard with (10)		Hard with (11)	
	$n=$ 100	$n=$ 500	$n=$ 100	$n=$ 500	$n=$ 100	$n=$ 500	$n=$ 100	$n=$ 500	$n=$ 100	$n=$ 500	$n=$ 100	$n=$ 500
dp	6	187	6	194	5	189	6	202	13	440	13	456
bb	16	30	18	36	19	308	24	67	1488	2585∗	36	∗
cplex	20	34	26	28	20	29	24	29	28	57	24	47
profitu10	16	27	15	21	12	20	19	22	12	41	18	31
profitu20	16	25	16	22	14	25	18	24	15	43	16	33
profitu30	12	25	16	23	13	27	18	26	12	52	15	41
envu10	15	21	13	22	13	28	17	22	16	39	17	32
envu20	14	25	13	23	12	26	14	26	19	46	15	34
envu30	14	24	13	21	11	27	15	26	22	56	14	42
envuinc10	16	22	14	19	12	22	17	21	16	37	17	44
envuinc20	15	22	13	22	11	27	14	25	15	48	15	37
envuinc30	15	22	13	22	12	29	15	24	16	58	17	51

∗ Indicates that there are instances that require more than 300 seconds to solve. The average time is calculated without these instances.

1299 and $c = 0.1 \times \sum_{i=1}^{n} w_i$. Also for (10) and (11), we have $p_1 \in [150, 250]$ for $n = 100$ and $p_1 \in [600, 700]$ for $n = 500$.

During the test on hard cases with formula (10), we found some special instances that CPLEX takes extremely long time, more than 20 seconds, to solve comparing with general instances, where the solving time is less than 0.1 seconds. We also tested our method on these instances.

5.2. *Experimental Results*

We tested our proposed approach for each case discussed above. The dynamic programming algorithm was applied to a subset of the reordered items (stages) and inequalities were obtained and added to the original integer programming formulation to solve the UKP. For comparison, classic dynamic programming approach and basic branch and bound approach are applied. We tested five random instances for each case. All experiments were conducted on a personal computer running Windows 7 with a 2.60 GHz CPU and 8.0 GB memory. And CPLEX 12.6 was applied.

We included the following models in experiments and compared the computational efficiency of solving each of them:
dp: Dynamic programming approach.
bb: Branch and bound approach.
cplex: Formulation (1) solved by CPLEX (default settings).
profitu: Valid inequalities (4) together with *cplex*.
envu: Concave envelope inequalities (6) derived in the last dynamic programming stage together with *cplex*.
envuinc: Concave envelope inequalities (6), which are derived through every stage, added incrementally to the standard formulation (1).

The integer number after models *profitu*, *envu* and *envuinc* represents the number of dynamic programming stages executed in the model.

Table 1 summarizes average time consumption for each model on each testing case except the hard case with (9). This type of problem is also known as subset sum problem

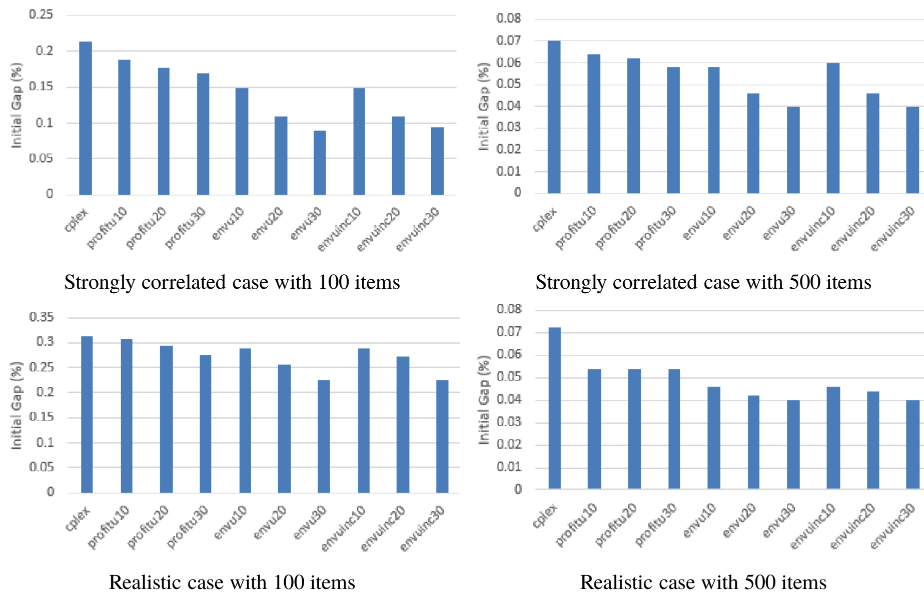

Fig. 4. Initial gap of *cplex* and proposed models for partial testing cases.

and can be solved by all models in the experiment in a short time. Result shows that *dp* approach is very sensitive to the size of the problem, both the number of items and the capacity of a knapsack. When there are 100 items with relatively small knapsack capacity, it has the best performance over all other models. However, when the number of items increases (as the result of how we decide capacity value, the capacity increases as well), the time spent on finding optimal solution increased dramatically. The testing result also shows the advantage of *dp* approach, i.e. computation times are stable for fixed size of problems. For *bb* approach, it is sensitive to the input parameters, such as the relationship between items' profits and weights. It has long computation times for hard cases with equation (10) and (11). For *cplex*, although it also tends to have longer time when problem size grows, unlike *dp*, the time increases moderately. Our proposed models have good performance on average. For most of cases tested, they solve problems faster than *bb* and *cplex*. And compared to *dp*, they are less sensitive to the size of problem.

Besides computation time, initial gap is another important measurement. Initial gap presents the relative difference between the first integer solution explored and the best upper bound then. A smaller initial gap implies that tighter lower bound or/and tighter upper bound has been found to facilitate pruning search space, eliminating candidate nodes. Therefore, usually less time is spent on exploring nodes of the search tree. For brevity, we only present the initial gap of *cplex* and our proposed models in a strongly correlated case and a realistic case in Fig. 4. Result demonstrates the dynamic-programming-based inequalities help to improve the initial gap in most of the experimental cases. The *envuinc* model, which is close to *envu* model but derives inequalities incrementally for each stage, has similar testing result to that of *envu*.

Table 2
Special instances solution time (in seconds).

Instance	cplex	profitu	envu	envuinc
1	22.45	0.07–5.51	0.07–5.47	0.05–2.05
2	42.18	0.10–0.11	0.06–0.12	0.07–0.09
3	72.84	0.05–0.08	0.07–0.21	0.08–0.16
4	145.95	0.06–0.07	0.07–0.11	0.10–0.20
5	188.87	0.23–1.87	0.23–1.07	0.15–0.24
6	5086.51	0.04–0.07	0.05–0.18	0.07–0.08
7	8095.91	0.19–0.30	0.25–81.66	0.23–26.96

Table 2 illustrates time consumption of *cplex* model and our proposed models on solving special instances found during instances generation of hard cases. With valid inequalities added to original integer programming formulation, these problems can be solved in a very short time.

Note that our experimental results indicate that examining more stages of the dynamic programming does not always provide better performance. Because there is a potential trade-off between obtaining computational advantage from a better initial solution caused by trimming model with more dynamic-programming-based inequalities and computational cost on calculating these inequalities.

6. Multi-Dimensional Unbounded Knapsack Problem

Most researchers consider the multi-dimensional unbounded knapsack problem as a special case of general integer problem with non-negative coefficients. There is little literature studying this problem based on the concept of knapsack. However, by embedding the notion of knapsack, our approach for single constraint unbounded knapsack problem could be extended and modified to solve this type of problems.

6.1. *Dominance Relations*

The concept of dominance relations was originally developed for the UKP. However, it could be extended to the d-UKP easily with the consideration of d attribute measurements, instead of one weight measurement, for each item type. In the d-UKP, item type j simply dominates item type k, if $p_j \geqslant p_k$ and $w_{ij} \leqslant w_{ik}$, $\forall i$. We could define multiple dominance, collective dominance and threshold dominance for the d-UKP in the same way. As d, the number of constraints in problem, increases, the efficiency and benefits of examining items' dominance relations decrease. Because, firstly, more attribute measurements need to be checked to determine whether they satisfy the requirement of dominance relation, which is time consuming. Taking detection of simple dominance relation as an example, the time needed to compare attributes' values among all pairs of items is $O(dn^2)$, which increases linearly with the number of constraints. Secondly, there will be less dominance relation among items due to more dominance requirements to qualify. In the case of randomly generated coefficients, for two randomly chosen items, the

Table 3
Percentage of simply dominated items in randomly generated cases.

	$d = 1$	$d = 2$	$d = 5$	$d = 10$	$d = 15$	$d = 20$
$n = 100$	95.8	87.2	42.7	7.3	0.2	0.0
$n = 500$	99.3	96.4	67.6	11.4	0.7	0.0

probability that one simply dominates the other is $(0.5)^d$, which decrease dramatically as d increases. Moreover, a test on instances with random coefficients has been conducted to investigate the percentage of simply dominated items with different numbers of constraints. The result is given in Table 3. If there are two constraints, more than 80% of items are dominated and could be removed from later consideration. Thus, the pre-processing of dominance relation is efficient in shrinking the size of problem. But as d increases, the number/percentage of simply dominated items will reduce, hence the size of problem couldn't be reduced much by pre-fixing variables of dominated items to zero. When problem has more than 15 constraints, pre-processing is not beneficial.

In general, with large d, more time is spent on dominance detection, but less dominated items could be found and removed from problem. Therefore, when it comes to the d-UKP, we only detect simple dominance relation among items for problems with no more than 10 constraints.

6.2. *Dynamic Programming with Lists*

For the UKP, we filled up a $c \times n$ table to store states. If the same dynamic programming approach is applied on the d-UKP, time and space required will be $O(mc_1c_2 \cdots c_d)$, which is computationally expensive even for a medium scale problem with a small number of constraints. To deal with this issue, list representation is used.

Because the dynamic programming function of the d-UKP is a step function, similar to that of the UKP, list could be utilized solely to contain states where the dynamic programming function value changes. This idea was applied for single constraint 0–1 knapsack problems in Horowitz and Sahni (1974) and multi-dimensional 0–1 knapsack problems in Balev *et al.* (2008). The following is our proposed generalization of dynamic programming with lists for the d-UKP with accommodation to inequalities derivation process.

During the dynamic programming process, a list of states will be created for each stage. Every state (β, f) represents a feasible solution, where β is capacity consumption in d dimensions and f is corresponding profit. To seek computational efficiency and ease of generating valid inequalities in next step, states in lists are required to be stored in the order of increasing total profit.

The first state list, L_1, is formed with all feasible non-negative integer multipliers of item type 1, i.e.

$$L_1 = \{(a \times \bar{W}_1, a \times p_1)\}, \quad a = 0, 1, \ldots, \qquad \min_{j=1\ldots d} \lfloor c_j / w_{j1} \rfloor,$$

where $\bar{W}_i$ is column vector $(w_{1i}, w_{2i}, \ldots, w_{di})$.

For other stages, the generation of L_t ($t \geqslant 2$) is based on states in L_{t-1}. Given a state (β_k, f_k) in L_{t-1}, new states added to L_t are

$$\{(\beta_k + a \times \bar{W}_t, f_k + a \times p_t)\},$$

where multiplier a is any non-negative integer number and new state is kept feasible. However, with this state generation scheme, an ordered list may not be obtained directly. To have an ordered list, a sorting process is required which will cost $O(mlogm)$, where m is the length of the list. To achieve computational efficiency, instead of sorting after the list is created, we suggest keeping an ordered list all the time during state generation with assistance of two pointers P_1 and P_2. P_1 indicates the state that currently is in use to create new state, while P_2 points to the location where the new generated state should be inserted into the list regarding its profit. Both pointers are initialized at the beginning of the list. New state generated is $(\beta_k + \bar{W}_t, f_k + p_t)$, where (β_k, f_k) is the state pointed by P_1. Since states in a list are ordered according to their profits, to find the correct place to insert new state, P_2 needs to move down through the list until the first state with higher profit comparing with new state's profit is found. Then the new state is inserted in front of the state labelled by P_2. Afterwards, P_1 is moved to the next state and this procedure is reposted until P_1 is at the end of the list and no more feasible state could be produced. Because both pointers only go through the list once, the complexity of set up list L_t is $O(m)$.

Similar to the concept of dominance relations among different item types, there are states considered to be non-promising during dynamic programming process, i.e. state (β_k, f_k) is a non-promising state if, for some other state (β_t, f_t), $\beta_k \geqslant \beta_t$ and $f_k \leqslant f_t$. These states won't lead to optimal solution, therefore they could be removed from a list to keep lists short and reduce the computation time and memory space requirement.

6.3. *Valid Inequalities Derivation*

We proposed two types of valid inequalities for the UKP in Section 4. Both of them could be applied to the d-UKP.

First, to derive inequality (4), the upper bound of total profits for the first i types of items is needed. Since we have ordered lists of states, the upper bound is the profit associated with the last state of L_i.

Second, to create inequality (6) for the d-UKP, the upper concave envelope of points in $(d + 1)$ dimensional space needs to be calculated instead of that in 2 dimensional space for the UKP (Figs. 5 and 6 show points and upper concave envelope for a problem with two constraints). Each attribute represents one dimension and total profit represents one additional dimension.

Obtaining the exact upper concave envelope as valid inequalities is computationally expensive even for low-dimensional problems. To simplify the process, single attribute dimension will be considered at a time. In other words, envelope is projected to the plane spanned by selected attribute and total profit (see Fig. 7). Then inequalities could be constructed with the same process as that for the UKP. This is repeated for all other attributes.

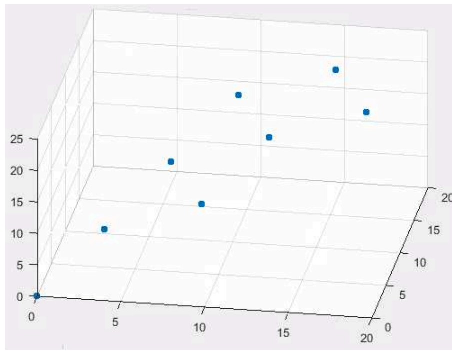

Fig. 5. Feasible states for a 2-d knapsack problem.

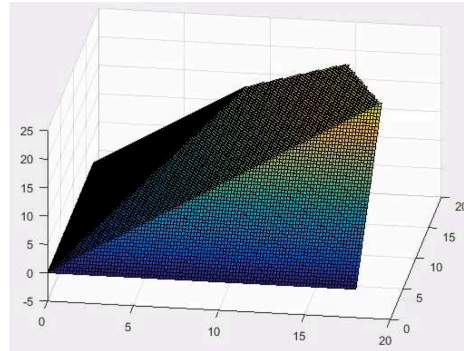

Fig. 6. Upper concave envelope of points in Fig. 5.

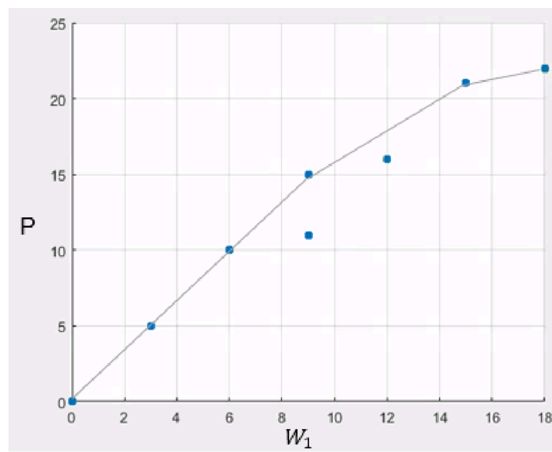

Fig. 7. Projection of points to 2-d plane and the upper concave envelope.

6.4. *Computational Results*

Our approach is tested on random cases of the d-UKP with 100 and 500 item types and dimension 2, 3, 5, 10 and 15 respectively. Profit and weights of each item type are uniformly generated from range $(1, 100)$. Capacity limits for all attributes are fixed to 300. Average results over 5 iterations are shown in Table 4.

Results indicate that our proposed method for the d-UKP helps to enhance efficiency by reducing the initial gap and the computation times for a medium-size problem. However, as the number of constraints grows, the advantage of this method becomes less obvious. First, as we mentioned in the previous section, when d becomes larger, fewer items are dominated by others. Therefore, fewer variables could be fixed during the pre-processing step. Second, dynamic programming process takes more time to be implemented. Third, the second type of inequalities for the d-UKP is obtained by relaxing $d - 1$ dimensions of the exact envelope, hence these inequalities become less accurate in defining the feasible region when d is a large number.

Table 4
Random cases for the d-UKP.

n	*d*	initgap (%)		Time (ms)		Nodes	
		cplex	ours	cplex	ours	cplex	ours
100	2	2.32	1.94	147	31.67	0	0
	3	4.16	3.90	165.33	31.33	0	0
	5	9.17	8.22	191	57	22	0
	10	24.34	14.40	287.67	188.33	771.67	747.33
	15	20.87	19.84	410.33	322.33	3833.3	3295
500	2	1.46	0.84	175	54	0	0
	3	3.09	1.29	208	54	0	0
	5	8.15	5.59	180	73	0	0
	10	27.1	14.37	1445	1006	11395	9520
	15	25.62	21.18	1718	1358	14006	10883

7. Conclusions

In this paper, we present a new approach to solve the UKP and the d-UKP to optimality by taking advantage of both dynamic programming algorithm and traditional integer programming algorithm. According to the information obtained from the partial execution of a dynamic programming recursion, we derive valid inequalities (4) and (6), which help to tighten the integer programming formulation and obtain better initial bounds of optimal solution. Therefore, after adding the dynamic-programming-based inequalities, search space is pruned more efficiently and the time spent on exploring the solution tree is reduced.

Acknowledgement. The authors would like to thank the referees for their valuable suggestions that improved the paper. Research was partially supported by the Paul and Heidi Brown Preeminent Professorship at University of Florida.

References

Andonov, R., Poirriez, V., Rajopadhye, S. (2000). Unbounded knapsack problem: dynamic programming revisited. *European Journal of Operational Research*, 123, 394–407.

Balev, S., Yanev, N., Freville, A., Andonov, R. (2008). A dynamic programming based reduction procedure for the multidimensional 0–1 knapsack problem. *European Journal of Operation Research*, 186, 63–76.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton.

Cabot, A.V. (1970). An enumeration algorithm for knapsack problem. *Operations Research*, 18, 306–311.

Dantzig, G.B. (1957). Discrete variable extremum problems. *Operations Research*, 5, 266–277.

Floudas, C.A., Pardalos, P.M. (2009). *Encyclopedia of Optimization*, 2nd edition. Springer, New York.

Garfinkel, R.S., Nemhauser, G.L. (1972). *Integer Programming*. Wiley, New York.

Gilmore, P.C., Gomory, R.E. (1963). A linear programming approach to the cutting stock problem – part. *Operations Research*, 11, 863–888.

Gilmore, P.C., Gomory, R.E. (1966). The theory and computation of knapsack functions. *Operations Research*, 14, 1045–1074.

Hartman, J.C., Buyuktahtakin, I.E., Smith, J.C. (2010). Dynamic-programming-based inequalities for the capacitated lot-sizing problem. *IIE Transactions*, 42, 915–930.

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

Lueker, G.S. (1975). *Two NP-complete problems in nonnegative integer programming*. Report No. 178. Computer Science Laboratory, Princeton University, Princeton.

Martello, S., Toth, P. (1984). A mixture of dynamic programming and branch-and-bound for the subset-sum problem. *Management Science*, 30, 765–771.

Martello, S., Toth, P. (1990a). An exact algorithm for large unbounded knapsack problems. *Operations Research Letters*, 9, 15–20.

Martello, S., Toth, P. (1990b). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, New York.

Poirriez, V., Yanev, N., Andonov, R. (2009). A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization*, 6, 110–124.

Sinha, A., Zoltners, A.A. (1979). The multiple-choice knapsack problem. *Operations Research*, 27, 503–515.

Yanasse, H.H., Soma, N.Y. (1987). A new enumeration scheme for the knapsack problem. *Discrete Applied Mathematics*, 18, 235–245.

X. He is currently a PhD candidate in the Department of Industrial and Systems Engineering, University of Florida. Her research interests are integer programming and dynamic programming.

J.C. Hartman was appointed Dean of the Francis College of Engineering at the University of Massachusetts Lowell in July of 2013. Prior to this he was Chairman of Industrial and Systems Engineering at the University of Florida, where he currently has a courtesy appointment. His research and teaching interests are in the areas of applied optimization, most notably dynamic programming, with applications in engineering economic decision analysis, transportation logistics and manufacturing logistics.

P.M. Pardalos serves as Distinguished Professor of Industrial and Systems Engineering at the University of Florida. Additionally, he is the Paul and Heidi Brown Preeminent Professor in Industrial and Systems Engineering. He is also an affiliated faculty member of the Computer and Information Science Department, the Hellenic Studies Center, and the Biomedical Engineering Department. He is also the Director of the Center for Applied Optimization. Dr. Pardalos is a world leading expert in global and combinatorial optimization. His recent research interests include network design problems, optimization in telecommunications, e-commerce, data mining, biomedical applications, and massive computing.

Dinaminiu programavimu pagrįstos nelygybės neapribotam sveikaskaitiniam kuprinės uždaviniui spręsti

Xueqi HE, Joseph C. HARTMAN, Panos M. PARDALOS

Siūlome naują hibridinį būdą spręsti neapribotam sveikaskaitiniam kuprinės uždaviniui, kur galiojančios nelygybės yra sugeneruotos iš tarpinių ekvivalenčios dinaminio programavimo formuluotės sprendinių. Šios nelygybės padeda sugriežtinti pirmines tiesiniu programavimu pagrįstas uždavinio relaksacijas ir pagerinti sprendimo skaičiuojamąjį efektyvumą. Taip pat šis būdas patobulinamas spręsti daugiamačiam neapribotam kuprinės uždaviniui. Skaičiuojamieji rezultatai rodo šio būdo veiksmingumą abiem uždaviniams spręsti.