Journal of the Operational Research Society

ISSN: 0160-5682 (Print) 1476-9360 (Online) Journal homepage: https://www.tandfonline.com/loi/tjor20

An improved branch and bound algorithm for a strongly correlated unbounded knapsack problem

Y-J Seong, Y-G G, M-K Kang & C-W Kang

To cite this article: Y-J Seong, Y-G G, M-K Kang & C-W Kang (2004) An improved branch and bound algorithm for a strongly correlated unbounded knapsack problem, Journal of the Operational Research Society, 55:5, 547-552, DOI: 10.1057/palgrave.jors.2601698

To link to this article: https://doi.org/10.1057/palgrave.jors.2601698

Journal of the Operational Research Society (2004) 55, 547–552

www.palgrave-journals.com/jors

An improved branch and bound algorithm for a strongly correlated unbounded knapsack problem

Y-J Seong*, Y-G G, M-K Kang and C-W Kang

Hanyang University, South Korea

An unbounded knapsack problem (KP) was investigated that describes the loading of items into a knapsack with a finite capacity, W, so as to maximize the total value of the loaded items. There were n types of an infinite number of items, each type with a distinct weight and value. Exact branch and bound algorithms have been successfully applied previously to the unbounded KP, even when n and W were very large; however, the algorithms are not adequate when the weight and the value of the items are strongly correlated. An improved branching strategy is proposed that is less sensitive to such a correlation; it can therefore be used for both strongly correlated and uncorrelated problems.
Journal of the Operational Research Society (2004) **55,** 547–552. doi:10.1057/palgrave.jors.2601698

Keywords: knapsack problem; branch and bound; combinatorial optimization; exact algorithm

Introduction

A knapsack problem (KP) is a combinatorial optimization problem that describes the loading of items into a knapsack with a finite weight capacity so as to maximize the total value of the loaded items. The problem is formulated as follows.[1] We have a knapsack of capacity W, into which we may load n types of items. Each item of type i has a value v_i, a weight w_i, and there are b_i copies (W, n, w_i, v_i and b_i are all positive integers). x_i is the number of loaded type i items.

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leqslant W$$

$$0 \leqslant x_i \leqslant b_i, \quad x_i : \text{integer}, \; i = 1, 2, 3, \ldots, n$$

Without loss of generality, we can assume that[2]

$$\frac{v_1}{w_1} \geqslant \frac{v_2}{w_2} \geqslant \cdots \geqslant \frac{v_n}{w_n} \tag{1}$$

When for all i, b_i is one, one or more, or infinite, the problem is termed a 0–1 KP, a bounded KP (BKP), or an unbounded KP (UKP), respectively.[2]

This paper concentrates on the UKP. The UKP is an integer KP or cargo-loading problem,[1,3] and is a classic NP-hard[4] and simple integer programming problem. It has a wide variety of both theoretical and practical application areas; therefore, many exact algorithms have been developed.

Dynamic programming is the classic approach for solving an UKP.[1,2,5,6] The time and space complexity are $O(nW)$ and $O(W)$, respectively. However, there is no guarantee that W is a polynomial function of n; therefore, dynamic programming is not adequate for an UKP when W and n are very large.[3]

Branch and bound algorithms based on a depth-first search method are usually used for a large UKP. Gilmore and Gomory,[7] Cabot,[8] Martello and Toth,[3] and others have proposed efficient branch and bound algorithms. The MTU (MTU1 or MTU2) by Martello and Toth is known to be the most efficient algorithm for the UKP.[3] However, the MTU requires a long time to find the optimal solution for a strongly correlated UKP (SCUKP) in which w_i and v_i are strongly correlated. This is especially true when v_i of each i corresponds to w_i plus a fixed constant, even if the problem is not large.[9] This is because the number of the non-dominated item types is not small in the SCUKP.

This paper proposes an improved branch and bound algorithm for the SCUKP, and the algorithm is comprised of a new branching strategy and the MTU1. The efficiency of the new algorithm is demonstrated through computational experiments.

The Martello and Toth branch and bound algorithm (MTU1)

The MTU uses a depth-first search method that is based on developing and backtracking phases. The MTU1 is

*Correspondence: Y-J Seong, Department of Industrial Engineering, Hanyang University, 1271 Sa-1 dong, Sangnok-gu, Ansan, Kyunggi-do 425-791, South Korea.
E-mail: ydudwh@empal.com

described in this section; the MTU2, an improvement to MTU1, is described in the next section.

In the developing phase, the number of loaded items is determined as follows. Suppose that the loaded item type and the residual capacity of the knapsack at some node N are i and W', respectively. Find a minimum j such that $j>i$ and $w_j \leqslant W'$. At the child node of N, the number of loaded type j items, x_j, is set to $\lfloor W'/w_j \rfloor$, and x_k such that $i<k<j$ is set to 0 ($\lfloor x \rfloor$ is the largest integer not greater than x). The residual capacity of the knapsack is reduced to $W'-w_jx_j$. If there exists no j such that $j>i$ and $w_j \leqslant W'$, then the total sum of the loaded item values is computed and the algorithm backtracks. If the total sum is greater than the current best solution value, the current best solution and its value are updated.

In the backtracking phase, one is subtracted from the number of loaded items. Suppose that the loaded item type and the residual capacity of the knapsack at some node N are j and W'. The maximum i is found such that $i \leqslant j$ and $x_i>0$. One is subtracted from x_i, and W' is set to $W'+w_i$. The algorithm then continues with the developing phase. If there exists no i such that $i \leqslant j$ and $x_i>0$, the algorithm stops.

The MTU1 is comprised of several efficient ideas that can improve the algorithm performance.[3] First, the algorithm contains an upper bound at each node. There is no need to branch at a node when the upper bound does not dominate the current best solution value. This eliminates unnecessary branching effort. The upper bound U is as follows. If W is the initial knapsack capacity and $\lceil x \rceil$ is the smallest integer not less than x, then

$$W' = W - \lfloor W/w_1 \rfloor w_1$$

$$z' = \lfloor W/w_1 \rfloor v_1 + \lfloor W'/w_2 \rfloor v_2$$

$$W'' = W' - \lfloor W'/w_2 \rfloor w_2$$

$$U' = z' + \left\lfloor W'' \frac{v_3}{w_3} \right\rfloor$$

$$U'' = z' + \left\lfloor \left(W'' + \left\lceil \frac{1}{w_1}(w_2 - W'') \right\rceil w_1 \right) \frac{v_2}{w_2} - \left\lceil \frac{1}{w_1}(w_2 - W'') \right\rceil v_1 \right\rfloor$$

Then,

$$U = \max(U', U'')$$

is an upper bound for the UKP.

Suppose that at some node N, the number of loaded type i items and the residual capacity of the knapsack are x_i and W', respectively, and there exists minimum j such that $j>i$ and $w_j \leqslant W'$. A new UKP can be created in which the capacity of the knapsack is W' and the types of items are $j, j+1, \ldots, n$. Suppose that the upper bound of the new UKP is $U(j, W')$ and the total sum of the values of the loaded items up to node N is $V(N)$. Then, the node N upper bound $U(N)$ equals $V(N)+U(j, W')$. Of course, $U(j, W')$ can be computed based on the same procedure defined for U. In this way, an upper bound can be computed for each node, and branching at that node can be eliminated when the upper bound cannot dominate the current best solution value.

The second efficient idea is applied to the backtracking phases. After subtracting 1 from x_i, if the trivial upper bound $V(N)+\lfloor W'(v_{i+1}/w_{i+1}) \rfloor$ is less than the current best solution value, x_i is set to 0 and the algorithm continues with the developing phase. This is because all of the upper bounds of the child nodes of N with $x_i>0$ are less than the current best solution value. The backtracking is repeated until x_i is reduced to 0. This last idea is also applied to the backtracking phase. It is used when the residual capacity, before subtracting 1 from x_i, is not sufficient for only one item of type j such that $j>i$.

The Martello and Toth *core problem* (MTU2)

From the experimental results obtained using the MTU1 algorithm, Martello and Toth identified the following behaviours:[3] (i) many examples of the UKP can be solved exactly within reasonable computing times, even for very large values of n; (ii) when a solution is possible, the sorting time required to satisfy inequality (1) is usually a large fraction of the total computing time; however, (iii) only the item types with the highest ratios of v_i/w_i are selected for the solution, that is, $\max\{j: x_j>0\} \ll n$. Therefore, they supposed that $v_i/w_i \geqslant v_{i+1}/w_{i+1}$ for $i=1, 2, \ldots, n-1$, and defined the *core* as

$$\boldsymbol{C} = \{1, 2, 3, \ldots, \bar{n} \equiv \max\{j : x_j>0\}\}$$

and the *core problem* as

$$\begin{aligned} &\text{Maximize} && \sum_{i \in c} v_i x_i \\ &\text{subject to} && \sum_{i \in c} w_i x_i \leqslant W \end{aligned}$$

$$x_i \geqslant 0 \text{ and integer}, \quad i \in \boldsymbol{C}.$$

If the value of $\bar{n}$ is known *a priori*, the UKP solution can be obtained by solving the *core problem* using the MTU1 and by setting $x_i=0$ for all i such that $i \notin \boldsymbol{C}$. Since $\bar{n}$ cannot be identified *a priori*, they determined an approximate core without sorting, as follows.

Assuming no condition on the ratio v_i/w_i, a tentative value of $\bar{n}$ is selected and the corresponding core problem is solved. If the solution value dominates an upper bound $U(j)$ for all j, then the solution is the optimum; otherwise, the value of $\bar{n}$ is

increased and the process is repeated until the optimum solution is obtained or the size of the core $\bar{n}$ equals n. $U(j)$ denotes the upper bound U with the additional constraint $x_j = 1$, that is, an upper bound for the UKP solution value if only one item of type j is used in the solution. The initial value of $\bar{n}$, ∂, was determined experimentally by Martello and Toth[3] as

$$\partial = \max(100, \lfloor n/100 \rfloor)$$

The kth largest ratio v_j/w_j was determined using the algorithm given in Fischetti and Martello,[10] the complexity of which is O(n).

Martello and Toth proved that given any example of an UKP and an item type k, if there exists an item type j such that

$$\left\lfloor \frac{w_k}{w_j} \right\rfloor v_j \geqslant v_k \tag{2}$$

then k is dominated and eliminated from the UKP without changing the optimal solution. Through experiments, they showed that the MTU2, which uses the *core problem*, is better than the MTU1.[3]

A new branching strategy

A new efficient branching strategy is proposed that can reduce repeated branching efforts when a branch and bound algorithm based on a depth-first search method is used for solving the UKP. The strategy is an improvement to the MTU1 and incorporates the same ideas. A new property is added that dramatically reduces the number of nodes visited.

Property 1 *Suppose that N_{new} is a new branching node and N_{old} is a branched node at which the residual capacity of the knapsack and the loaded item type is the same as those at N_{new}. If there exists N_{old} such that $V(N_{\text{old}}) \geqslant V(N_{\text{new}})$, then all of the branched solutions at N_{new} are not better than the current best solution.*

Proof Suppose that the loaded item type is i and the residual capacity of the knapsack is W' at some node N_{new}, and that the item type and residual capacity of the knapsack at node N_{old} are also i and W', respectively. The upper bounds at node N_{new} and N_{old} are $V(N_{\text{new}}) + U(j, W')$ and $V(N_{\text{old}}) + U(j, W')$, where j is the minimum k such that $k > i$ and $w_k \leqslant W'$. By supposition,

$$V(N_{\text{old}}) \geqslant V(N_{\text{new}})$$

Then,

$$V(N_{\text{old}}) + U(j, W') \geqslant V(N_{\text{new}}) + U(j, W')$$

Therefore, Property 1 holds. □

Property 1 requires memory space for i, W', and $V(N)$ that are obtained at the visited nodes. Thus, a matrix $\boldsymbol{M}$ is defined:

$$\boldsymbol{M} = \begin{bmatrix} M_{1,1} & M_{1,2} & \cdots & M_{1,W} \\ M_{2,1} & M_{2,2} & \cdots & M_{2,W} \\ \vdots & M_{i,W'} & \cdots & \vdots \\ M_{n',1} & M_{n',2} & \cdots & M_{n',W} \end{bmatrix} \tag{3}$$

where n' is the non-dominated number of items and $M_{i,W'}$ is the maximum $V(N_{i,W'})$. $N_{i,W'}$ indicates the nodes at which the loaded item type is i and the residual capacity of the knapsack is W'. Therefore, when branching at a node N at which the loaded item type is i and the residual capacity of the knapsack is W', if $V(N) > M_{i,W'}$, then the algorithm continues to branch at N after $M_{i,W'}$ has been updated with $V(N)$; otherwise, the algorithm stops branching and backtracks.

Since $\boldsymbol{M}$ tends to be large for a SCUKP in which n and W are also large, a data structure is used that can allocate the elements of matrix $\boldsymbol{M}$ to memory space dynamically. The initial size of $\boldsymbol{M}$ is 0, and a new element of $\boldsymbol{M}$ is dynamically allocated to memory space only when the element does not already exist in $\boldsymbol{M}$. Therefore, $\boldsymbol{M}$ is usually much smaller than the number of visited nodes; in the worst case, the size of $\boldsymbol{M}$ reaches the number of visited nodes. Dynamic programming requires constant memory space in proportion to the capacity W. However, the branch and bound algorithm with $\boldsymbol{M}$ requires only a small amount of memory, although there is no guarantee that the size of $\boldsymbol{M}$ is a polynomial function of n only.

Computational experiments indicated that when the dominated items were eliminated, the total computing time was reduced significantly for all cases of a SCUKP with a large value of n. According to inequality (2), the dominated items are eliminated as follows.

Procedure 1: Eliminating the dominated items

```
N = {1, 2, …, n}
For j = 1 to |N| – 1
  For k = j + 1 to |N|
    If ⌊w_k/w_j⌋ v_j ≥ v_k then
      N = N – {k}
    Else if ⌊w_j/w_k⌋ v_k ≥ v_j then
      N = N – {j}, k = |N|
    End if
  End for
End for
```

After eliminating the dominated items, $\boldsymbol{N} = \{1, 2, \ldots, n'\}$ and usually $n' \ll n$. The proposed algorithm is summarized as follows, where $\hat{\boldsymbol{x}}$, $\boldsymbol{x}$ and $\hat{z}$ describe the current best solution, current feasible solution, and current best solution value, respectively.

Procedure 2: Proposed algorithm

Step 1. [Initialize]

Eliminate dominated items according to Procedure 1.
Sort the non-dominated items according to decreasing v_i/w_i ratios.

$\hat{\boldsymbol{x}} = \boldsymbol{0}$, $\boldsymbol{x} = \boldsymbol{0}$, $i = 1$, $\hat{z} = 0$

Initialize empty sparse matrix $\boldsymbol{M}$.
$x_1 = \lfloor W/w_1 \rfloor$, $V(N) = v_1 x_1$, $W' = W - w_1 x_1$
Calculate U.
Find $m_i = \min\{w_j: j > i\}$ for all $i = 1, 2, \ldots, n'$.

Step 2. [Develop]

If $W' < m_i$ then
  If $\hat{z} < V(N)$ then
    $\hat{z} = V(N)$, $\hat{\boldsymbol{x}} = \boldsymbol{x}$
    If $\hat{z} = U$ then go to Step 5.
  End if
  Go to Step 3.
Else
  Find min j such that $j > i$ and $w_j \leqslant W'$.
  If $V(N) + U(j, W') \leqslant \hat{z}$ then go to Step 3.
  If $M_{i,W'} \geqslant V(N)$ then go to Step 3.
  $x_j = \lfloor W'/w_j \rfloor$, $V(N) = V(N) + v_j x_j$, $W' = W' - w_j x_j$
  $M_{i,W'}$ $V(N)$, $i = j$
  Go to Step 2.
End if

Step 3. [Backtrack]

Find max j such that $j \leqslant i$ and $x_j > 0$.
If $j < 1$ then go to Step 5.
$i = j$, $x_i = x_i - 1$
$V(N) = V(N) - v_i$, $W' = W' + w_i$
If $W' < m_i$ then go to Step 3.
If $V(N) + \left\lfloor W' \frac{v_{i+1}}{w_{i+1}} \right\rfloor \leqslant \hat{z}$ then
  $V(N) = V(N) - v_i x_i$, $W' = W' + w_i x_i$, $x_i = 0$
  Go to Step 3.
End if
If $W' - w_i \geqslant m_i$ then go to Step 2.

Step 4. [Replace a jth item with an hth item]

$j = i$, $h = j + 1$
If $\hat{z} \geqslant V(N) + \left\lfloor W' \frac{v_h}{w_h} \right\rfloor$ then go to Step 3.
If $w_h \geqslant w_j$ then
  If $w_h = w_j$ or $w_h > W'$ or $\hat{z} \geqslant V(N) + v_h$ then
    $h = h + 1$
    Go to Step 4.
  End if $\hat{z} = V(N) + v_h$, $\hat{\boldsymbol{x}} = \boldsymbol{x}$, $x_h = 1$
  If $\hat{z} = U$ then go to Step 5.
  $j = h$, $h = h + 1$
  Go to Step 4.
Else
  If $W' - w_h < m_{h-1}$ then
    $h = h + 1$
    Go to Step 4.
  End if
  $i = h$, $x_i = \lfloor W'/w_i \rfloor$
  $V(N) = V(N) + v_i x_i$, $W' = W' - w_i x_i$,
  Go to Step 2.
End if

Step 5. [Finish]

Exit with $\hat{z}$ and $\hat{\boldsymbol{x}}$.

Computational results and analysis

Martello and Toth[3] used data sets to describe the UKP, where the number of item types, n is very large:

uncorrelated: v_i and w_i uniformly random in [10, 1000];
strongly correlated: w_i uniformly random in [10, 1000],

$$v_i = w_i + 100.$$

For all cases, W was set to $0.5 \sum_{i=1}^{n} w_i$ for $n \leqslant 100\,000$, and to $0.1 \sum_{i=1}^{n} w_i$ for $n > 100\,000$ to avoid integer overflows. But the range [10, 1000] is so narrow that it generates, on the average, only about 1000 distinct item types when $n = 100\,000$, each of which has about 100 instances of the same weight and value. Thus, the data sets were modified to avoid trivial occurrences:

n: 10 000, 20 000,…, 100 000;
uncorrelated: v_i and w_i uniformly random in [100, 10n];
strongly correlated: w_i uniformly random in [100, 10n],

$$v_i = w_i + 1000.$$

A total of 100 problems were generated for each n, and W was set to $0.1 \sum_{i=1}^{n} w_i$. The broadened range [100, 10n] enabled the generated items to have diverse weights and values. The lower bound was set to 100 to improve the validity of the tests (Martello and Toth used 10 at the lower bound in their experiments). The number of non-dominated items cannot outnumber the smallest w_i in the SCUKP, in which v_i of each i corresponds to w_i plus a fixed constant. In the SCUKP, when the lower bound of w_i is set to a small number, the number of non-dominated items is also small. Thus, the dominating time is a very large fraction of the total computing time and the main algorithm contributes little to the overall performance.

The experiments were run on a 32-bit IBM compatible PC with a 650-MHz CPU and 128 MB RAM. The code was written using C++. All times reported are in CPU-seconds. For comparison purposes, the MTU2 code was also programmed in C++ and run on the same PC. Table 1 shows the experimental results of 100 cases for each n with strongly correlated data sets. The proposed algorithm was faster than the MTU2 for all cases, and solved large problems in a short time. For larger values of n, the ratio of

computing time between the proposed algorithm and MTU2 was large.

The number of non-dominated items, n', tends to become large in proportion to n. This is one of the principal reasons why strongly correlated problems are difficult to solve. The MTU2 usually requires a computing time of less than 1 s to obtain a solution, but it is known to take over 100 s to solve certain problems. The computing times shown for the MTU2 in Table 1 are large, especially for large values of n (or n'). This anomaly makes it difficult to forecast the computing time of the MTU2 for SCUKPs. The proposed algorithm, however, required a more consistent amount of computing time and this is because of property 1, which can reduce repeated branching efforts in branch and bound algorithm.

Table 2 shows average and maximum sizes of the matrix $\boldsymbol{M}$ for the same data sets shown in Table 1. The size of $\boldsymbol{M}$ increased in proportion to n (or n'), but the rate of increase was small compared to that of n. The proposed algorithm required a relatively small amount of memory to store $\boldsymbol{M}$ for a large SCUKP. A binary search tree data structure was used, so that only 24 bytes were required for each element. Therefore, the amount of memory required to store the largest $\boldsymbol{M}$ (11,461 elements) was only 275 kilobytes.

Table 3 shows the results of 100 cases for each n with uncorrelated data sets; v_i and w_i were uniformly random in [100, 10n]. The other experimental conditions were the same as given in Table 1. The proposed algorithm was always faster than the MTU2 algorithm, and reduced the computing time by up to 64%. Thus, the proposed algorithm can also be applied successfully to uncorrelated UKPs. In uncorrelated UKPs, a small number of item types with large v_i/w_i ratios dominated the other item types, which explains why n' is small compared to the values given in Table 1, and does not increase in proportion to n.

Conclusion

A new branch and bound algorithm was proposed for a strongly correlated UKP. The algorithm uses an efficient branching strategy that eliminates unnecessary branching efforts. The experimental results showed that the proposed algorithm significantly reduced the computing time for strongly correlated and uncorrelated problems. The proposed algorithm was more effective when the problem was very large or the weight and value of the items were strongly correlated. Moreover, the algorithm required a relatively constant amount of computing time, regardless of the problem. Therefore, the proposed algorithm can be used to solve all types of UKPs, regardless of the data correlation.

Table 2 Size of matrix $\boldsymbol{M}$

n	*Average size of n'*	*Average size of $\boldsymbol{M}$ used*	*Largest size of $\boldsymbol{M}$ used*
10 000	27	628	5539
20 000	46	1251	7142
30 000	60	1817	8976
40 000	70	2218	9340
50 000	77	2464	10 155
60 000	83	2983	9850
70 000	87	3395	10 303
80 000	91	3048	11 242
90 000	93	3401	10 963
100 000	95	3195	11 461

Table 3 Computational results for uncorrelated UKPs

	Proposed		*MTU2*
n	*Average size of n'*	*Average computing times*	*Average computing times*
10 000	3	0.009	0.025
20 000	4	0.019	0.051
30 000	4	0.027	0.078
40 000	4	0.037	0.104
50 000	4	0.046	0.130
60 000	4	0.056	0.155
70 000	4	0.064	0.184
80 000	4	0.075	0.209
90 000	4	0.084	0.234
100 000	4	0.093	0.263

Table 1 Computational results for SCUKPs

	Proposed		*MTU2*	
n	*Average size of n'*	*Average computing time*	*Average computing time*	*Number of anomaly**
10 000	27	0.013	0.055	(0,5)
20 000	46	0.027	0.182	(0,10)
30 000	60	0.042	0.960	(2,15)
40 000	70	0.056	1.501	(2,20)
50 000	77	0.068	3.367	(4,45)
60 000	83	0.084	5.190	(6,30)
70 000	87	0.097	8.495	(7,35)
80 000	91	0.103	10.100	(9,40)
90 000	93	0.117	11.388	(9,45)
100 000	95	0.125	12.391	(8,50)

*(m,s) indicates that m problem instances took more than s seconds.

References

1 Dreyfus SE and Law AM (1977). *The Art and Theory of Dynamic Programming*. Academic Press, Inc.: Orlando.
2 Nemhauser GL and Wolsey LA (1988). *Integer and Combinatorial Optimization*. John Wiley & Sons: New York.
3 Martello S and Toth P (1990). *Knapsack Problem: Algorithms and Computer Implementations*. John Wiley & Sons: Chichester.
4 Andonov R, Poirriez V and Rajopadhye S (2000). Unbounded knapsack problem: dynamic programming revisited. *Eur J Opl Res* **123**: 394–407.
5 Hansen P and Ryan J (1996). Testing integer knapsacks for feasibility. *Eur J Opl Res* **88**: 578–582.
6 Winston WL (1991). *Introduction to Mathematical Programming Applications & Algorithms*. PWS-KENT Publishing Company: Boston.
7 Gilmore PC and Gomory RE (1966). The theory and computation of knapsack functions. *Opns Res* **14**: 1045–1074.
8 Cabot AV (1970). An enumeration algorithm for knapsack problems. *Opns Res* **18**: 306–311.
9 Pisinger D (1998). A fast algorithm for strongly correlated knapsack problems. *Discrete Appl Math* **89**: 197–212.
10 Fischetti M and Martello S (1988). A hybrid algorithm for finding the kth smallest of n elements in O(n) time. *Ann Ops Res* **13**: 401–419.

Received January 2002;
accepted November 2003 after one revision