ELSEVIER

Discrete Applied Mathematics 77 (1997) 271–280

DISCRETE APPLIED MATHEMATICS

Simple but efficient approaches for the collapsing knapsack problem [☆]

Ulrich Pferschy[a,*], David Pisinger[b], Gerhard J. Woeginger[c]

[a] Universität Graz, Institut für Statistik und Operations Research, Universitätsstr. 15, A-8010 Graz, Austria
[b] DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark
[c] TU Graz, Institut für Mathematik B, Steyrergasse 30, A-8010 Graz, Austria

Received 3 July 1995; revised 2 May 1996

Abstract

The collapsing knapsack problem is a generalization of the ordinary knapsack problem, where the knapsack capacity is a non-increasing function of the number of items included. Whereas previous papers on the topic have applied quite involved techniques, the current paper presents and analyzes two rather simple approaches: One approach that is based on the reduction to a standard knapsack problem, and another approach that is based on a simple dynamic programming recursion. Both algorithms have pseudo-polynomial solution times, guaranteeing reasonable solution times for moderate coefficient sizes. Computational experiments are provided to expose the efficiency of the two approaches compared to previous algorithms.

Keywords: Collapsing knapsack problem; Nonlinear Knapsack; 0–1 Programming

1. Introduction

The classical *knapsack problem* tries to find a maximal-valued subset of items with a limited total *weight*. An extension of this problem arises if the capacity of the knapsack depends on the number of items it contains. The resulting 0–1 *collapsing knapsack problem* is defined as

$$\text{(CKP)}\quad \text{maximize} \quad \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_i x_i \leqslant b\left(\sum_{i=1}^{n} x_i\right), \quad x_i \in \{0, 1\}, i = 1, \ldots, n,$$

[☆] This research was partially supported by the Spezialforschungsbereich F003 "Optimierung und Kontrolle", Projektbereich Diskrete Optimierung, and by the Christian Doppler Laboratorium für Diskrete Optimierung.

* Corresponding author.E-mail:pferschy@kfunigraz.ac.at.

where c_i resp. $a_i \in \mathbb{N}^+$ are the value resp. weight of item i and $b(i)$ is a *nonincreasing* functional over $\{1, \ldots, n\}$ representing the capacity of the knapsack. This means that the capacity of the knapsack will decrease the more items are packed into the knapsack. If $b(\cdot)$ is a linear function, the problem becomes equivalent to a standard knapsack problem.

The collapsing knapsack problem was introduced by Posner and Guignard [7] and has applications in satellite communication where transmissions on the band require gaps between the portions of the band assigned to each user. Other applications are time-sharing computer systems, where each process run on the system causes additional overhead, and design of a shopping center where both the type and size of a shop must be selected [7].

In the case of b being a *nondecreasing* functional, the problem can be seen as a 0–1 *expanding knapsack problem*. It can be interpreted as a "rubber knapsack" whose capacity increases the more items it contains. Like the collapsing knapsack problem it has several applications, e.g. in budget control, where the dealer gives trade discount depending on the number of items purchased, or in manpower planning of state subsidized projects, where the size of a grant depends on the number of employees.

The collapsing knapsack problem is $\mathcal{NP}$-hard as it contains the 0–1 knapsack problem as a special case. The present paper shows that it may be solved in pseudo-polynomial time through dynamic programming. The first implicit enumeration algorithm for CKP was given by Posner and Guignard [7], but recently Fayard and Plateau [2] presented a superior algorithm using new tight upper and lower bounds.

Although the standard knapsack problem with fixed capacity can be solved very efficiently, currently known CKP algorithms cannot guarantee to solve instances of larger size (e.g. $n \geqslant 100$) to optimality within reasonable time. To build a bridge between these two related problems, we show in Section 2 that any 0–1 collapsing knapsack problem can be represented as an equivalent *standard* 0–1 *knapsack problem* (SKP) with a fixed capacity. The resulting SKP belongs to the class of "hard" knapsack instances because the values and weights of the items are highly correlated. For these kinds of problem instances special algorithms have been developed by e.g. Martello and Toth [3] and Pisinger [6]. We briefly discuss the practical aspects of solving the SKP at the end of Section 2.

Several solution techniques known from classical knapsack problems can also be applied directly to CKP. In particular, a *dynamic programming* approach for CKP is presented in Section 3, showing that the problem may be solved in time $O(n^2 b(1))$. Some simple upper bound tests are introduced to limit the enumeration further.

These two new approaches are finally compared to the implicit enumeration algorithm by Fayard and Plateau [2] in Section 4. A study of the solution times based on data models from the literature shows that the reduction to SKP is an attractive alternative for small instances. The reduction is easy to implement and the resulting SKP can be solved by standard knapsack routines which are present in many computational environments. Thus, hardly any new code has to be written to solve small problems.

However, the dynamic programming approach, which is also easy to handle, turned out to be *superior for all instances* considered and should be applied for all medium and large problems.

2. Reduction to a standard knapsack problem

We describe a reduction scheme which generates for any given collapsing knapsack problem CKP an equivalent instance of a standard knapsack problem SKP. In fact, the reduction is valid for *arbitrary* capacity function $b(\cdot)$, since we do not make use of the monotonicity.

The described construction can be performed in $O(n)$ time and leads to an instance of SKP that is twice as large as the original instance of CKP. Further properties and solution methods for the resulting SKP are given at the end of this section.

Let $A = \sum_{i=1}^{n} a_i$ and $C = \sum_{i=1}^{n} c_i$. Without loss of generality, we may assume that $0 \leqslant b(i) \leqslant A$ for all i, as otherwise we replace $b(i)$ by A. Moreover, we may assume $n \geqslant 3$. We construct an instance of the standard 0–1 knapsack problem with $2n$ items having weights $\alpha_1, \ldots, \alpha_{2n}$ and values $\gamma_1, \ldots, \gamma_{2n}$. The weights are defined as

$$\alpha_i = \begin{cases} a_i + A & \text{for } i = 1, \ldots, n, \\ (4n - i)A - b(i - n), & \text{for } i = n + 1, \ldots, 2n, \end{cases}$$

while the corresponding values of the new items are defined by

$$\gamma_i = \begin{cases} c_i + C & \text{for } i = 1, \ldots, n, \\ (3n + 1 - i)C, & \text{for } i = n + 1, \ldots, 2n. \end{cases}$$

Finally, the size B of the standard knapsack is $3nA$.

The problem SKP consists in finding a subset of the items with maximum value and weight at most B, thus

$$\text{(SKP) maximize} \quad \sum_{i=1}^{2n} \gamma_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{2n} \alpha_i x_i \leqslant B, \quad x_i \in \{0, 1\}, i = 1, \ldots, 2n.$$

Items with index in $\{1, \ldots, n\}$ are called *small items* and items with index in $\{n + 1, \ldots, 2n\}$ are called *large items*.

Lemma 1. *In a feasible solution of the above instance of SKP, the knapsack contains at most one large item. In case it does contain a large item with index j, $n + 1 \leqslant j \leqslant 2n$, it contains at most $j - n$ small items*

Proof. The weight α_j of any large item with index $n+1 \leqslant j \leqslant 2n$ is at least

$$(4n-j)A - b(j-n) \geqslant 2nA - A \geqslant (2n-1)A$$

and at most $(3n-1)A$. Hence, no two large items fit together into the knapsack but every single large item fits.

For the second statement in the lemma, assume there is a large item with index $n+1 \leqslant j \leqslant 2n$ in the knapsack. For the small items, there remains a weight capacity of $(j-n)A + b(j-n)$. Since every item has weight greater than A and since $b(j-n) \leqslant A$, at most $j-n$ small items can be packed. □

Lemma 2. *In a feasible solution of the above instance of SKP with objective value at least* $(2n+1)C+1$, *the knapsack contains a large item with index* $n+1 \leqslant j \leqslant 2n$ *and exactly* $j-n$ *small items.*

Proof. The overall value of all small items is $(n+1)C$, hence there must be some large item j in the knapsack. Let I denote the set of indices of small items in the knapsack. Suppose $|I| \leqslant j-n-1$. Then the overall value of the knapsack is at most

$$\gamma_j + \sum_{i\in I}(C+c_i) = (3n+1-j)C + |I|C + \sum_{i\in I} c_i \leqslant (2n+1)C,$$

a contradiction. Hence, there are at least $j-n$ small items in the knapsack and the statement in Lemma 1 completes the argument. □

Theorem 3. *The instance of CKP has a feasible solution with objective value* V *if and only if the instance of SKP has a feasible solution with objective value* $V+(2n+1)C$.

Proof. (*Only if*) Assume CKP has a feasible solution with objective value V and define the set $I = \{i \mid x_i = 1\}$. In other words, $\sum_{i\in I} c_i = V$ and $\sum_{i\in I} a_i \leqslant b(|I|)$ holds. Define a solution for SKP that contains all small items with indices in I and the large item with index $n+|I|$. The objective value of this solution is exactly $V+(2n+1)C$, and the weight is

$$\alpha_{n+|I|} + \sum_{i\in I}\alpha_i = (3n-|I|)A - b(|I|) + \sum_{i\in I}(A+a_i) = 3nA$$
$$+ \left(\sum_{i\in I} a_i - b(I)\right) \leqslant 3nA = B.$$

Hence, we have found a feasible solution for SKP with objective value $V+(2n+1)C$.

(*If*) Now assume that SKP has a feasible solution with objective value $V+(2n+1)C$. Lemma 2 states that the knapsack contains a large item with index $n+1 \leqslant j \leqslant 2n$ and exactly $j-n$ small items. Let I denote the set of indices of small items in the knapsack, $|I| = j-n$. The overall weight of a feasible solution for SKP is at most B, i.e.

$$B = 3nA \geqslant \alpha_j + \sum_{i \in I} \alpha_i = (4n - j)A - b(j - n) + \sum_{i \in I} (A + a_i) = 3nA$$
$$+ \left(\sum_{i \in I} a_i - b(|I|) \right).$$

Hence, $\sum_{i \in I} a_i \leqslant b(|I|)$. The objective value of the solution is

$$(2n + 1)C + V = \gamma_j + \sum_{i \in I} \gamma_i = (3n + 1 - j)C + \sum_{i \in I} (C + c_i) = (2n + 1)C + \sum_{i \in I} c_i.$$

Consequently, $\sum_{i \in I} c_i = V$. Summarizing, the items of CKP with indices in I have objective value V and weight at most $b(|I|)$. □

There is one disadvantage of the above reduction scheme: very large coefficients appear, and moreover the values and weights are highly correlated. Both properties make these instances difficult to solve.

Martello and Toth [3] consider the so-called *uncorrelated instances with similar weights*, where the items are randomly generated as a_j randomly distributed in [100 000, 100 100] while the values c_j are distributed in [1, 1000]. It is demonstrated that such instances are very hard to solve. The SKP instances considered in this paper, however, are even more difficult as the values are also distributed in a large interval and have a large correlation with the weights.

It should thus be expected that SKP is nearly impossibly to solve. This was indeed the case when we tried to solve these instances using the MT2 algorithm by Martello and Toth [4], as only very small sized instances could be solved in reasonable time. Thus, we tried instead to solve SKP by using the MINKNAP algorithm presented in [6]. The MINKNAP algorithm is based on dynamic programming, thus having a time bound $O(nB)$, which in our case also corresponds to $O(n^2A)$ as $B = 3nA$. If we consider instances where the weights are bounded by a fixed constant a', the solution time becomes $O(n^3a')$ which however for moderate weight values and moderate problem sizes is practicable. Detailed results on the computational experiments are given in Section 4.

3. A dynamic programming approach to CKP

A simple dynamic programming algorithm (DCKP) with time bound $O(n^2b')$ where $b' = \max_{i=1,\dots,n} b(i)$ may be defined as follows: Let $f_{i,k}(\tilde{c})$; $i = 1, \dots, n$; $k = 1, \dots, i$; $\tilde{c} = 0, \dots, b(k)$ be an optimal solution to the following two-constrained knapsack problem defined on the first i items:

$$f_{i,k}(\tilde{c}) = \max \left\{ \sum_{j=1}^{i} c_j x_j \;\middle|\; \sum_{j=1}^{i} a_j x_j \leqslant \tilde{c};\ \sum_{j=1}^{i} x_j = k;\ x_j \in \{0, 1\}, j = 1, \dots, i \right\}.$$

Initially, we set $f_{0,0}(\tilde{c}) = 0$ for all $\tilde{c} = 0, \dots, b'$, and $f_{0,k}(\tilde{c}) = -\infty$ for all $k = 1, \dots, n$ and all $\tilde{c} = 0, \dots, b'$. Subsequent values of $f_{i,k}$ are easily found by using the recursion

$$f_{i,k}(\tilde{c}) = \max\begin{cases} f_{i-1,k}(\tilde{c}) \\ f_{i-1,k-1}(\tilde{c} - a_i) + c_i & \text{if } k > 0, \tilde{c} - a_i \geqslant 0. \end{cases}$$

At any step i of the recursion, we may derive a lower bound z as

$$z = \max_{k=0,\dots,i} f_{i,k}(b(i)),$$

which for $i = n$ yields the optimal objective value. The corresponding solution vector may be found by backtracking through the states at previous levels of the recursion. Note that the recursion actually does not make use of the monotonicity of $b(\cdot)$. Thus, both the dynamic programming scheme and the reduction of Section 2 are valid for arbitrary distributions of the capacity function.

If we use dynamic programming by reaching, we only need to save states (π, μ) with $\pi = f_{i,k}(\mu)$, thus generally decreasing the time and space demand considerably. A further improvement is obtained by using some bounding rules to fathom inferior states. For this purpose we assume that the capacities are in nonincreasing order. Without loss of generality, we may also sort the items according to nonincreasing value-to-weight ratio. Then we have

Proposition 4. *An upper bound on a state (π, μ) where $\pi = f_{i,k}(\mu)$ is given by*

$$u(\pi, \mu) = \lfloor \pi + (b(k+1) - \mu)c_{i+1}/a_{i+1} \rfloor.$$

Proof. As per definition the state (π, μ) corresponds to a solution vector with $\sum_{j=1}^{i} x_j = k$, thus any improved solution must contain $k+1$ or more items. Then the bound appears by linear relaxation as a solution to: $\max\{\pi + \sum_{j=k+1}^{n} c_j x_j \mid \mu + \sum_{j=k+1}^{n} a_j x_j \leqslant b(k+1);\ x_j \geqslant 0\}$. $\square$

Thus we may fathom any state with $u(\pi, \mu) \leqslant z$ as it will not lead to an improved solution. The following proposition further reduces the number of states we have to consider.

Proposition 5. *Let $h = \max\{j \mid \sum_{i=1}^{j} a_i \leqslant b(j)\}$ and define $K = \max\{j \mid b(j) \geqslant \sum_{i=1}^{h} a_i\}$. Then $\sum_{i=1}^{n} x_i \leqslant K$ in any optimal solution to CKP.*

Proof. Let $B = \sum_{i=1}^{h} a_i$. An optimal solution to the 0–1 knapsack problem: $\max\{z = \sum_{i=1}^{n} c_i x_i \mid \sum_{i=1}^{n} a_i x_i \leqslant B;\ x_i \in \{0, 1\}\}$ is given by $z = \sum_{i=1}^{h} c_i$, which also is a feasible solution to CKP. For any 0–1 knapsack problem defined on the same items but with smaller capacity $b(k) < B$, $k = K+1, \dots, n$ and additional constraint $\sum_{i=1}^{n} x_i = k$, no better objective value is obtainable than z. $\square$

Hence, we can ignore all states $\pi = f_{i,k}(\mu)$ with $k > K$, meaning that the time complexity becomes $O(nKb(1))$.

Further improvements may be obtained by deriving upper and lower bounds on the number of items in an optimal solution as well as deriving tighter bounds in the enumeration. Both techniques are described in [2], but we chose not to incorporate these reductions in order to keep things simple.

4. Computational experiments

We compare the two presented simple approaches for the CKP with the algorithm FPCK90 by Fayard and Plateau [2]. The latter is a very advanced algorithm, deriving upper and lower bounds by outer linearization of the capacities $b(\cdot)$, as well as using some specialized reduction rules for the CKP. As the code for FPCK90 is not available the solution times in this section have been taken from [2].

The instances considered are generated as described in [2, 7]: Values c_j are randomly distributed in $[1, c']$, the weights a_j are distributed in $[1, a']$ while the capacities $b(\cdot)$ are constructed as follows: Generate m random numbers in $[1, b']$, sort these values in nonincreasing order and assign them to $b(1), \ldots, b(m)$, setting $b(j) = 0$ for $j = m + 1, \ldots, n$.

The computational experiments by Fayard and Plateau [2] were carried out on a IBM 9370/60, while the algorithms presented here were tested on a Digital AlphaServer 2000 5/250 with performance index 5.96 (SPECint95), resp. 8.39 (SPECfp95). Although our computational times are not directly comparable to the times given in [2], the characteristics of each of the algorithms are so striking, that we may disregard these differences.

Fayard and Plateau assigned an upper limit of 45 min for the solution of each instance, while we use a space bound of 32 Mb for each of the algorithms. In those cases where not all of the ten examples generated for each parameter set could be solved within these bounds the entry in the table is empty.

Table 1 considers small-sized instances with $n = 30$. All the algorithms are able to solve these instances in fractions of a second, and this also applies to Table 2 where instances of size $n = 50$ are considered.

Medium-sized instances with $n = 100$ are considered in Table 3. The FPCK90 code is not able to solve several of these instances, for large values of b'. On the other hand both of the algorithms presented here are able to solve the instances in reasonable time, although the dynamic programming algorithm clearly is superior to the SKP algorithm.

Finally, Table 4 considers large sized instances with $n = 1000$. The FPCK90 algorithm is only able to solve these instances if the number of items in the knapsack is very small. On the other hand the SKP algorithm has better solution times the more items fit into the knapsack. This may be explained by the well-known fact [1], that it is relatively easy to obtain a filled knapsack if many items are present, and thus to obtain a good lower bound. Instances with large values of m could however not be tested for

Table 1
Solution times in seconds, small problems $n = 30$, average of 10 instances

n	c'	a'	b'	m	FPCK90	SKP	DCKP
30	100	100	300	10	0.02	< 0.01	< 0.01
				20	0.03	0.01	< 0.01
				30	0.01	0.01	< 0.01
30	100	100	600	10	0.32	< 0.01	< 0.01
				20	0.84	0.01	< 0.01
				30	0.19	0.01	< 0.01
30	200	100	300	10	0.01	< 0.01	< 0.01
				20	0.01	0.01	< 0.01
				30	0.02	0.01	< 0.01
30	200	100	600	10	0.03	< 0.01	< 0.01
				20	0.05	0.01	< 0.01
				30	0.03	0.01	< 0.01

Table 2
Solution times in seconds, small problems $n = 50$, average of 10 instances

n	c'	a'	b'	m	FPCK90	SKP	DCKP
50	300	1000	1000	10	0.03	0.01	< 0.01
				20	0.02	0.06	< 0.01
				30	0.03	0.13	< 0.01
				40	0.03	0.19	< 0.01
				50	0.03	0.19	< 0.01
50	300	1000	2000	10	0.06	0.01	< 0.01
				20	0.04	0.05	< 0.01
				30	0.14	0.12	< 0.01
				40	0.03	0.17	< 0.01
				50	0.04	0.19	< 0.01

the SKP algorithm, due to the space limit. Finally, we notice that the dynamic programming algorithm has superior solution times even for these large sized instances, solving all problems in less than 1 minute.

From these experiments we may conclude, that for instances up to medium size, the SKP algorithm is a good alternative to FPCK90, due to its stable behavior. None of the two algorithms is however able to solve large-sized instances when the range of $b(\cdot)$ gets large. These instances may only be solved by the dynamic programming algorithm, which also for the small-sized instances is superior to both of the previous approaches. The dynamic programming recursion is however very simple, but by using some improved reduction rules on an expanding core problem (see e.g. [5] for details) we may expect to solve even very large-sized collapsing knapsack problems in reasonable time.

Table 3
Solution times in seconds, medium-sized problems $n = 100$, average of 10 instances

n	c'	a'	b'	m	FPCK90	SKP	DCKP
100	300	1000	1000	10	0.03	0.05	< 0.01
				30	0.26	0.64	< 0.01
				50	0.13	1.84	< 0.01
				70	0.04	3.30	< 0.01
				100	0.22	4.57	< 0.01
100	300	1000	5000	10	—	0.05	< 0.01
				30	20.78	0.60	< 0.01
				50	—	1.67	< 0.01
				70	—	2.94	< 0.01
				100	148.23	3.15	< 0.01
100	300	1000	10000	10	—	0.05	< 0.01
				30	—	0.53	0.01
				50	—	1.46	0.01
				70	—	2.69	0.01
				100	—	2.47	0.01

Table 4
Solution times in seconds, large-sized problems $n = 1000$, average of 10 instances

n	c'	a'	b'	m	FPCK90	SKP	DCKP
1000	300	1000	1000	100	187.83	3353.47	0.03
				500	0.42	—	0.03
1000	300	1000	10000	100	—	1387.76	0.97
				500	—	—	0.95
1000	300	1000	50000	100	—	938.18	16.10
				500	—	—	39.94

It is also important to emphasize the importance of having pseudo-polynomial time bounds for the two presented algorithms, as this guarantees reasonable and more or less predictable solution times for moderate coefficient sizes. In contrast to this, the FPCK90 algorithm has exponentially growing solution times in the worst case, although it may be quite fast for some instances. The overall performance of FPCK90 seems to be very unstable for large problems as can be seen in Tables 3 and 4.

Acknowledgements

The authors would like to thank Gérard Plateau for pointing out the problem during the ESI-X summer school at Groupe HEC, Paris.

References

[1] E. Balas and E. Zemel, An algorithm for large zero–one knapsack problems, Oper. Res. 28 (1980) 1130–1154.

[2] D. Fayard and G. Plateau, An exact algorithm for the 0–1 collapsing knapsack problem, Discrete Appl. Math. 49 (1994), 175–187.

[3] S. Martello and P. Toth, Upper bounds and algorithms for hard 0–1 knapsack problems, (1996) Oper. Res., to appear.

[4] S. Martello and P. Toth, A new algorithm for the 0–1 knapsack problem, Management Sci. 34 (1988) 633–644.

[5] D. Pisinger, A minimal algorithm for the multiple-choice knapsack problem, European J. Oper. Res. 83 (1995) 394–410.

[6] D. Pisinger, A minimal algorithm for the 0–1 knapsack problem, Oper. Res., (1996) to appear.

[7] M.E. Posner and M. Guignard, The collapsing 0–1 knapsack problem, Math. Programming 15 (1978) 155–161.