How to solve the collapsing subset-sum problem revisited

Iida, Hiroshi*

Abstract

This is a revised version of Iida [5]: We introduce a new type of problem that we shall call collapsing subset-sum problem, and present an algorithm to solve the problem. The problem is a special case of the collapsing knapsack problem, and the algorithm based on a depth-first branch-and-bound strategy, involving some tip, makes it easy to solve the problem.

Keywords: Collapsing knapsack problem; Branch-and-bound; Depth-first search; Strongly correlated knapsack problem; Subset-sum problem

MSC 2010: 90C57; 90C09; 90C27

1 Introduction

In the classical 0–1 knapsack problem (hereafter KP), given a knapsack of a certain capacity and items each of which has profit and weight, we pack the items into the knapsack so that the total profit of packed items is maximised without the total weight of those exceeding the capacity. In the KP the capacity is a constant while Posner and Guignard [12] introduced a more complicated one with a nonconstant capacity, called the collapsing 0–1 knapsack problem, or CKP for short. As the name shows, the knapsack of CKP will collapse according to the number of packed items. For example, each item is a bottle, and should be wrapped up when packed. Then, the larger the number of packed items the smaller the capacity of the knapsack due to the wrapping of each item. For further applications, see [12]. The CKP is formulated as follows:

$$\begin{aligned} \text{(CKP)} \quad \text{maximise} \quad & \sum_{j\in N} c_j x_j, \\ \text{subject to} \quad & \sum_{j\in N} a_j x_j \le b\left(\sum_{j\in N} x_j\right), \\ & x_j \in \{0,1\},\ j \in N, \end{aligned} \tag{1}$$

where $N := \{1, 2, \ldots, n\}$, and each $j \in N$ indicates an item. The coefficients of profit c_j and weight a_j associated with any item j are given positive integers, and the $b(\cdot)$ is a given monotonically nonascending function on the discrete domain N as $b(1) \ge b(2) \ge \cdots \ge b(n)$. Also, the 0–1 variable x_j corresponds to the selection of item j as $x_j = 1(\text{packed})/0(\text{otherwise})$. Without loss of generality we assume that $a_j \le b(1)$ for any j, and $\sum_j a_j > b(n)$. Clearly, CKP is $\mathcal{NP}$-hard as an extension of KP. Incidentally, CKP itself has an extension called the continuous collapsing knapsack problem, see Posner and Suzuki [13].

Here we introduce several notation: Let $J \subseteq N$. Then we call the cardinality of J and $\sum_{j\in J} a_j$ the length and weight of J, respectively. In addition, let $x := (x_1, x_2, \ldots, x_n)$. Then

*E-mail: auau2.a.go.go@gmail.com

we call such an n-vector a solution, and naturally identify a solution x with a subset J as $x_j = 1 \Leftrightarrow j \in J$; hence, we sometimes call a subset of N a solution.

Now we consider applying an additional constraint to (1) such that $c_j = a_j$ for all $j \in N$ in the same way as producing the subset-sum problem (SSP) from KP. Then we have the following, named collapsing subset-sum problem (henceforth referred to as CSSP):

$$\begin{aligned} \text{(CSSP)} \quad \text{maximise} \quad & \sum_{j \in N} a_j x_j, \\ \text{subject to} \quad & \sum_{j \in N} a_j x_j \leq b\left(\sum_{j \in N} x_j\right), \\ & x_j \in \{0,1\},\ j \in N. \end{aligned}$$

This arises by way of example when there is just one kind of item; that is, the profit is directly proportional to its weight for any item. Indeed, for some $k > 0$ in the case of $c_j = ka_j$ for all j, the objective function of (1) turns out to be $k(\sum_j a_j x_j)$, and we shall maximise $\sum_j a_j x_j$.

Including SSP of $\mathcal{NP}$-hard as a special case, CSSP is still $\mathcal{NP}$-hard. Although well known that the additional constraint produces a hard instance of KP (see, e.g. Kellerer et al [8, p. 152]), is CSSP also hard to solve as an instance of CKP? In what follows we will survey studies to date on CKP, focusing on the case of CSSP.

In the literature several algorithms for CKP have so far been proposed, e.g., FPCK90 by Fayard and Plateau [3]. Since it would seem that the performance of the algorithm depends on whether profit to weight ratios $\{c_j/a_j\}_{j \in N}$ are widely distributed or not, FPCK90 appears not to be so promising enough to solve CSSP. To be more specific, first, three bounds concerning the cardinality of an optimum solution derived from Propositions 3.1–3.3 will work even for CSSP whereas Proposition 3.4[†] (resp. 3.5) shall deliver a trivial upper bound $\lfloor b(|B|) \rfloor$ (resp. $\lfloor b(|B|+1) \rfloor$) for the CSSP of $\sum_j x_j \geq |B|$ (resp. $\sum_j x_j \geq |B|+1$) like the linear programming relaxation applied to SSP or the multiple-choice subset-sum problem [8, Subsection 11.10.1]. Next, the outer-linearisation involved in FPCK90 approximates $b(i)$ as $b(i) \leq \bar{\alpha} \cdot i + \bar{\beta}$ with some appropriate $\bar{\alpha}(\leq 0), \bar{\beta}$, and generates an ordinary KP. The KP is utilised not only to obtain an upper bound analogous to the well-known Dantzig [1]'s upper bound in Proposition 3.7 but also to try to fix 0–1 variables with bounds analogous to those by Dembo and Hammer [2] in Proposition 3.8. In short, both the Dantzig's upper bound and the bounds by Dembo and Hammer don't seem so efficient for CSSP, because the resulting KP in the case of CSSP is identical with the strongly correlated 0–1 knapsack problem (SCKP) of a negative fixed-charge (i.e. $c_j = a_j + \bar{\alpha}$). Reflecting the very nature of SCKP, the performance of FPCK90 shall depend on the magnitude of $\bar{\alpha}$. The smaller the better. Indeed, it is well known that SCKP is generally not so easy to treat (see, e.g. [8, p. 151]).[‡] Further, SCKP arises as a result of applying inner-linearisation to given CSSP, too. A greedy algorithm applied to the SCKP gives a lower bound.

Also, Pferschy et al [10] proposed two algorithms for CKP, called SKP and DCKP respectively. The former transforms given CKP into an equivalent KP of $2n$ items, and solves the equivalent by

[†]According to Fayard and Plateau [3] their Proposition 3.4 is Theorem 1 in Posner and Guignard [12].

[‡]Consider the following instance of SCKP with fixed-charge -10 and capacity 130:

j	1	2	3	4	5	6
a_j	33	30	28	27	25	20

Here we will try to peg x_1 at 1. A solution (1 1 1 1 0 0) of profit sum 78 is greedily gained as $\sum_{j=1}^{4} a_j = 118 \leq 130 < 118 + \min\{a_5, a_6\}$. Solving a linear programming relaxation problem with $x_1 = 0$, we have $20 + 18 + 17 + 15 + 10 = 80 > 78$. Thus we could not fix even item 1 of the largest profit to weight ratio.

an algorithm for KP; The latter is based on the dynamic programming, involving the reduction of states. Focusing on the case of CSSP: On SKP, in a candidate for optimum packing besides one item $j\,(>n)$ we shall select $j-n$ item(s) among n items on each of which the profit is equal to the weight. Hence the resultant KP is nearly SSP. Further to the transformation of CKP into KP, even with an alternative [6] the obtained is nearly SCKP[§]; On DCKP, as to the upper bound $u(\pi,\mu) := \lfloor \pi + (b(k+1)-\mu)c_{i+1}/a_{i+1} \rfloor$ proposed in Proposition 4 so as to reduce states, the ratio c_{i+1}/a_{i+1} included is always equal to 1, and further it will not be rare for CSSP that π gets close to $\mu\,(\pi \le \mu)$, which makes the upper bound close to $\lfloor b(k+1) \rfloor$ quite natural as produced under $\sum_j x_j \ge k+1$. Moreover, in the case of CSSP, any order of items has fulfilled the assumption of $c_1/a_1 \ge c_2/a_2 \ge \cdots \ge c_n/a_n$. Therefore, the performance of the reduction of states by Proposition 5 shall depend on the order of items; however, there is no extra rule stated explicitly such as when $c_j/a_j = c_{j+1}/a_{j+1}$ the heavier precedes the other.

Furthermore, another algorithm for CKP was proposed by Wu and Srikanthan [15]. The algorithm would also seem to depend on the variety of profit to weight ratios. For example, both upper bounds u_k and u_r presented therein deliver the capacity $\lfloor b(j) \rfloor$ of no use for CSSP (The 2nd term in the definition of u_r [15, p. 1744] will be not $-\sum_{i=s+1}^{r} p_i$ but $-\sum_{i=s}^{r} p_i$). In addition, the proposed algorithm solves subproblems, in each of which a cardinality constraint is given, in nonascending order of the profit sum of their break solution; however, without sorting the subproblems, it will be sufficient for CSSP to solve the subproblems in ascending order of cardinality in the same way as an algorithm to be proposed in Section 2.

As observed in Pisinger [11] it seems that the direction of current study on KP is to develop a robust algorithm applicable to even an instance of various structures such as SSP, SCKP, and so on. Although challenging certainly, it will be rather down to earth for now to employ a tailored algorithm when a given problem has a special structure and there is an efficient algorithm exploiting the special structure. Indeed, both SSP and SCKP with huge coefficients are not yet tractable even for state-of-the-art methods for KP as seen in Tables 5.7–5.9 in [8] where the upper data ranges of those examined are 10^5, 10^6 and 10^7 (A single exception is the case where one based on a branch-and-bound strategy is applied to SSP. This fact is actually highly correlated with an algorithm that we are going to propose herein) whereas two algorithms specialised for SCKP, namely LSSCOR due to Pandit and Ravi Kumar [9] and XISC in [4], can manage instances with $n = 500$ and coefficients of value up to $2 \cdot 10^6$ as demonstrated in [4].

In the light of this it will be worthwhile to develop an algorithm specialised for CSSP. In the rest, Section 2 illustrates an algorithm for CSSP, and Section 3 presents brief computational experiments. The last section is devoted to conclusions.

2 An algorithm for CSSP

In fact, our algorithm is the aforementioned XISC with a minor modification. The XISC transforms given SCKP into equivalent SSPs with an additional constraint in the same way as LSSCOR:

$$\begin{aligned} \text{maximise} \quad & \sum_{j\in N} c_j x_j, \\ \text{subject to} \quad & \sum_{j\in N} c_j x_j \le \mathrm{MCL}(\beta) := b + \beta k,\ \beta = \sum_{j\in N} x_j, \\ & x_j \in \{0,1\},\ j \in N, \end{aligned} \tag{2}$$

[§]Albeit not elaborated here there is yet another transformation from CKP to KP by Wu et al [14], in which the capacity of KP yielded is $2(n+2)b(1)$ according to its summary whilst that of [6] can be estimated at $(2n+1)b(1)$.

where $c_j = a_j + k$ for all $j \in N$, and the b is a constant capacity. If the fixed-charge k is negative, MCL(β) is a monotonically descending function as to β. This means that (2) can be regarded as a special case of CSSP. Conversely, XISC in which MCL(β) is replaced with $b(\beta)$ is applicable to CSSP, because XISC does not depend on the linearity of MCL(β). Then a rough algorithmic sketch of XISC that we are going to propose for solving CSSP is as follows:

```
Sort all weights so that a_n ≥ a_{n-1} ≥ ··· ≥ a_1 ;
LB := max{i | Σ_{j=n-i+1}^{n} a_j ≤ b(i)} ; /* LB ≥ 1 */
z* := Σ_{j=n-LB+1}^{n} a_j ; /* initial incumbent value */
i := LB + 1 ;
while i ≤ n and z* < b(i) do /* main loop */
    Discard any a_j, provided a_j > b(i) − Σ_{l=1}^{i-1} a_l ;
    Enumerate all subsets of N with length i,
        updating z* with an improved one (if found);
    i := i + 1 ;
done.
```

Eventually, the final z^* is optimal. In the sketch, LB is a lower bound on the length of the subsets of N, and $\{n, n-1, \ldots, n-\mathrm{LB}+1\}$ gives maximum weight among any subset whose length is LB or less. Note that the loop starts at not $i = n$ but $i = \mathrm{LB}+1$, because the smaller i the more promising for optimality due to nonascending $b(i)$. A point is that if we found a subset of weight $b(i)$ while enumerating those of length i, then we could terminate the processing to optimality forthwith as labelled **exit** in a pseudo-code presented later. This is the crucial benefit of a branch-and-bound strategy as stated at the end of Subsection 5.5.2 in Kellerer et al [8].

In the following we show a pseudo-code of the main loop in the sketch. As seen in the pseudo-code, XISC is based on a depth-first search, and the root node of a *binary* search-tree corresponds to a subproblem in which $x_n, x_{n-1}, \ldots, x_{\mathrm{pos}+1}$ are all fixed at 0. We first explore a subtree arising by fixing $x_{\mathrm{pos}} = 1$, then explore the other with $x_{\mathrm{pos}} = 0$. For efficiency an array F is provided as $F[i] := \sum_{j=1}^{i} a_j \, (1 \le i < n)$.

```
while (i ≤ n and z* < b(i)) {
    Clear x_1, x_2, ..., x_n;
    pos := max{i, max{j | a_j + F[i − 1] ≤ b(i)}} ; /* pos ≥ i */
    j := pos; /* 'j' holds the latest selected item's index */
    x_j := 1; sum := a_j; rst := i − 1; /* rst + Σ_j x_j = i */
eval:
    while (true) {
        if (j − 1 = rst) { /* no more room for consideration */
            j := 1; Assign 1 to x_1, x_2, ..., x_rst; sum := sum + F[rst]; rst := 0;
            if (sum > b(i)) { skc := 1; goto backward; }
            if (sum > z*) {
                z* := sum;
                if (z* = b(i)) exit;
            }
            skc := 2; goto backward; /* skip */
        } else if (sum+F[rst] > b(i)) {
            ;  /* forward movement */
        } else if (sum + F[j − 1] − F[j − 1 − rst] ≤ z*) { /*  j − 1 > rst  */
            skc := 2; goto backward;
        } else if (rst = 1) { /* select the last item */
```

```
                    S := max_{1≤k<j}{k | sum + a_k ≤ b(i)}; /* NB. sum+a_1 ≤ b(i) */
                    if (sum + a_S > z*) {
                        z* := sum + a_S;
                        if (z* = b(i)) exit;
                    }
                    if (j − S ≤ 2) { skc := 3 − (j − S); goto backward; } /* skip */
                } else { /* rst > 1 */
                    j := j − 1; x_j := 1; sum := sum + a_j; rst := rst − 1; goto eval;
                }
forward:        x_j := 0; j := j − 1; x_j := 1; sum := sum − a_{j+1} + a_j;
            }
backward:   while (j ≤ pos and skc > 0) {
                skc := skc − (1 − x_j);
                if (x_j = 1) { x_j := 0; sum := sum − a_j; rst := rst + 1; }
                j := j + 1;
            }
            Find the smallest k ∈ [j, pos] for which x_k = 1; /* impossible if j > pos */
            if (found such k) { j := k; goto forward; }
            i := i + 1;
        }
```

Here we shall describe how 'skip' works. Note that when a solution of length i and weight less than $b(i)$ is obtained, we invoke skip. To take an example, consider an instance of CSSP with $n = 8$, and the processing is at $i = 4$. Recall that if our choice is represented by a bit pattern $(x_8x_7 \cdots x_1)_2$ then we enumerate subsets of length 4 from $(11110000)_2$ to $(00001111)_2$ in principle. Assume that we are now addressing a solution $J := \{7, 6, 4, 2\}$ of weight less than $b(4)$, so we would like to proceed as far as a place where we will obtain a solution heavier than J. At first glance, it may appear sufficient that we move to the root node of a subtree arising by fixing $x_4 = 0$; however, $\{7, 6, 3, 2\}$ is not heavier than J. In this case we should at least move to the root node of a subtree arising by fixing $x_6 = 0$. In another view, considering again that our choice is represented by a bit pattern, we attempt to find first '1' after skipping over two or more '0's to the left from a position indicating the latest selected item, then we replace the found '1' with '0', and leave all other bits to the right free; that is, we traverse the tree from $(01101010)_2$ to $(010\texttt{xxxxx})_2$, where the symbol `x` stands for x_j free—not yet fixed. Consequently, $\{7, 5, 4, 3\}$ may satisfy our demand. In short, to attain a solution heavier than a current one we have to skip over more than one '0' in a bit pattern corresponding to the current solution. To take another example under the same situation, if we invoke skip at $\{7, 6, 5, 2\}$—that is, $(01110010)_2$—then we move to $(0110\texttt{xxxx})_2$; thus, $\{7, 6, 4, 3\}$ may be heavier than $\{7, 6, 5, 2\}$.

3 Computational experiments

This section presents brief computational experiments. Provided data instances are the same as those of Tables 3 and 4 in Pferschy et al [10] (corresponding to Tables 5 and 6 in Fayard and Plateau [3]) except needless profits randomly generated in the range of $[1, 300]$. In Table 1, the pair of columns for b' and m is a basis to produce capacities: We randomly generate m integers in $[1, b']$, and assign them to $b(1)$ through $b(m)$ in nonascending order. For all $i > m$ we set $b(i) := 0$. Weights are randomly generated in $[1, 1000]$ for all cases except that, when $b(1) \leq 1000$, another range $[1, b(1) - 1]$ is used instead (This exceptional treatment to generate

weights is by ourselves only). Here, we did not adopt data instances in Wu and Srikanthan [15] although produced along the same lines as [3, 10] because all b' in Tables 3 and 4 thereof is 1000 equal to the upper data range of weights generated, which leads to that a few items have constructed an optimum solution to CSSP in most cases, which we ascertained by preliminary experiments, and such a situation is too favourable for XISC.

We implemented XISC in C, and did experiments on PowerBook with PowerPC G4 1.33 GHz and 512 Mb main memory. The XISC implemented has, within one millisecond, solved an instance determined by any triplet (n, b', m) in the left half of Table 1. Also, the columns for SKP and DCKP in Table 1 were quoted from Pferschy et al [10] as is for reference, where each entry represents average computing time of ten instances generated, or is —, indicating at least one instance among ten could not be solved within a space limit of 32 Mb on Digital AlphaServer 2000 5/250 referred therein. The last column of Table 1 is from Wu and Srikanthan [15] as is too (EXPCKP is the name of an algorithm proposed in [15]) where only an entry whose data type exists in Table 1 was cited, and the reference machine is one equipped with Celeron 300A.

Table 1: Examined data types and average computing time of ten instances by SKP, DCKP and EXPCKP, expressed in seconds

n	b'	m	SKP	DCKP	EXPCKP
100	1000	10	0.05	< 0.01	0.00
		30	0.64	< 0.01	
		50	1.84	< 0.01	
		70	3.30	< 0.01	0.00
		100	4.57	< 0.01	0.00
	5000	10	0.05	< 0.01	0.11
		30	0.60	< 0.01	
		50	1.67	< 0.01	
		70	2.94	< 0.01	0.65
		100	3.15	< 0.01	1.14
	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	
1000	1000	100	3353.47	0.03	1378.78
		500	—	0.03	
	10000	100	1387.76	0.97	
		500	—	0.95	
	50000	100	938.18	16.10	
		500	—	39.94	

It might seem not to be so meaningful to compare XISC with SKP, DCKP or EXPCKP directly, because XISC merely solves a special case of CKP while the others solve general CKP. However, as pointed out in Section 1, SKP, DCKP and EXPCKP appear to include some disadvantageous points on CSSP; therefore, when applied to CSSP they will not perform a lot better than the times shown in Table 1. Moreover, roughly speaking, our reference machine of a little bit outdated ought not to perform a thousand times faster than that of Pferschy et al [10] or Wu and Srikanthan [15].

Finally we would like to add that our experiments of $(n, b', m) = (1000, 10000, 500)$ showed that in most cases we had an optimum solution of, for some i, length i and weight $b(i)$. However, computing times for instances of $n = 1000$ are nearly the same irrespective of b' and m, which can also be found out in Table 1 of Iida [5, p. 145]. This will imply that, besides the benefit of a branch-and-bound strategy as stated before in relation to **exit**, the condition $z^* < b(i)$ at the top of the main loop works. As for our experiments of $b' = 1000$ in Table 1, indeed, an optimum solution often consists of a_n only, provided not $a_n < b(2)$. In the case of $a_n \geq b(2)$, the main loop starting at $i := \mathrm{LB} + 1\,(= 2)$ shall not be executed.

4 Conclusions

This paper has introduced the collapsing subset-sum problem, and—so as to solve it—has presented an algorithm based on an orthodox branch-and-bound strategy but involving a 'skip' facility, which has also been incorporated into algorithms for other knapsack-type problems, namely SSP [7] and SCKP [4]. The facility makes the implementation of a depth-first search light, because it does not require any calculation concerning weights for determining an appropriate position (node) to which we should move in a binary search-tree.

References

[1] Dantzig, G.B., April 1957, Discrete-variable extremum problems. *Operations Research* **5**(2) 266–277 [doi:10.1287/opre.5.2.266].

[2] Dembo, R.S. and Hammer, P.L., 1980, A reduction algorithm for knapsack problems. *Methods of Operations Research* **36**, 49–60.

[3] Fayard, D. and Plateau, G., 1994, An exact algorithm for the 0-1 collapsing knapsack problem. *Discrete Appl Math* **49**(1-3) 175–187 [doi:10.1016/0166-218X(94)90208-9].

[4] Iida, H., 1998, A simple branch-and-bound approach for the strongly correlated knapsack problem. *Economic Review: The bulletin of Otaru University of Commerce* **48**.2-3, 353–370; available as ⟨`http://hdl.handle.net/10252/866`⟩.

[5] Iida, H., 1998, How to solve the collapsing subset-sum problem. *Economic Review: The bulletin of Otaru University of Commerce* **48**.4, 141–146; available as ⟨`http://hdl.handle.net/10252/848`⟩.

[6] Iida, H. and Uno, T., 2002, A short note on the reducibility of the collapsing knapsack problem. *Journal of the Operations Research Society of Japan* **45**(3) 293–298.

[7] Iida, H. and Vlach, M., 1996, An exact algorithm for the subset-sum problem. In: *Cooperative Research Report* **92**, *Optimization — Modeling and Algorithms* — 9, 11–29 (Tokyo: Institute of Statistical Mathematics) available as ⟨`http://hdl.handle.net/10252/4041`⟩.

[8] Kellerer, H., Pferschy, U. and Pisinger, D., 2004, *Knapsack Problems* (UK: Springer).

[9] Pandit, S.N.N. and Ravi Kumar, M., 1993, A lexicographic search for strongly correlated 0-1 knapsack problems. *Opsearch* **30**(2) 97–116.

[10] Pferschy, U., Pisinger, D. and Woeginger, G.J., 1997, Simple but efficient approaches for the collapsing knapsack problem. *Discrete Applied Mathematics* **77**(3) 271–280 [doi:10.1016/S0166-218X(96)00134-5].

[11] Pisinger, D., 2005, Where are the hard knapsack problems? *Computers & Operations Research* **32**(9) 2271–2284 [doi:10.1016/j.cor.2004.03.002].

[12] Posner, M.E. and Guignard, M., 1978, The collapsing 0–1 knapsack problem. *Mathematical Programming* **15**(2) 155–161 [doi:10.1007/BF01609014].

[13] Posner, M.E. and Suzuki, H., 1987, A dual approach for the continuous collapsing knapsack problem. *Mathematical Programming* **39**(2) 207–214 [doi:10.1007/BF02592953].

[14] Wu, J., Lei, Y. and Schröder, H., 2001, A minimal reduction approach for the collapsing knapsack problem. *Computing and Informatics* **20**(4) 359–369.

[15] Wu, J. and Srikanthan, T., 2006, An efficient algorithm for the collapsing knapsack problem. *Information Sciences* **176**(12) 1739–1751 [doi:10.1016/j.ins.2005.07.014].