New trends in exact algorithms for the 0-1 knapsack problem

Silvano Martello[1], **David Pisinger**[2], **Paolo Toth**[1]
[1]DEIS, Univ. of Bologna, Viale Risorgimento 2, Bologna
[2]DIKU, Univ. of Copenhagen, Universitetsparken 1, Copenhagen

April 18, 1997

Abstract

While the 1980s were focused on the solution of large sized "easy" knapsack problems, this decade has brought several new algorithms, which are able to solve "hard" large sized instances. We will give an overview of the recent techniques for solving hard knapsack problems, with special emphasis on the addition of cardinality constraints, dynamic programming, and rudimentary divisibility. Computational results, comparing all recent algorithms, are presented.

1 Introduction

We consider the classical 0-1 Knapsack Problem (KP) where a subset of n given items has to be packed in a knapsack of capacity c. Each item has a profit p_j and a weight w_j and the problem is to select a subset of the items whose total weight does not exceed c and whose total profit is a maximum. We assume, without loss of generality, that all input data are positive integers. Introducing the binary decision variables x_j, with $x_j = 1$ if item j is selected, and $x_j = 0$ otherwise, we get the ILP-model:

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le c \\ & x_j \in \{0,1\}, \quad j \in \{1,\dots,n\}. \end{aligned} \tag{1}$$

[0]Technical Report 97/10, DIKU, University of Copenhagen, Denmark

The problem, which is $\mathcal{NP}$-hard, has been thoroughly studied in the last few decades and several exact algorithms for its solution can be found in the literature. If we were to give a brief description of the state of the art, we could say that

i) Problems with exponentially growing coefficients cannot be solved efficiently, and we should not expect them to be solved efficiently due to the $\mathcal{NP}$-hardness of the problem.

ii) Problems with bounded coefficients can however be solved very fast if the LP and the ILP solutions are sufficiently close to each other. This applies to instance types such as the so-called *uncorrelated*, *weakly correlated* and *subset-sum* problems.

iii) Problems with bounded coefficients where the LP solutions differ considerably from the ILP solutions are still difficult to solve by means of non-specialized algorithms. Among these instances we have the so-called *strongly correlated* problems and different variants. Besides, non-fill problems such as the *even-odd* problems are difficult to solve.

In the following we will give a historical overview of the different approaches, which have gradually extended the classes of instances that can be solved in a reasonably short time.

2 Basic branch-and-bound algorithms

Algorithms for knapsack problems are mainly based on two approaches: branch-and-bound and dynamic programming. The actual performance of an algorithm, however, strictly depends on the way tight upper bounds are applied.

The first upper bound for KP, based on a continuous relaxation, was presented by Dantzig [3] in the mid 1950s. It is obtained by sorting the items so that

$$\frac{p_j}{w_j} \geq \frac{p_{j+1}}{w_{j+1}} \qquad (j = 1, \ldots, n-1) \tag{2}$$

and determining the *critical variable* x_s through

$$s := \min\left\{ i \ : \textstyle\sum_{j=1}^{i} w_j > c \right\} \tag{3}$$

The continuous upper bound for KP is then

$$\left\lfloor \sum_{j=1}^{s-1} p_j + \left(c - \sum_{j=1}^{s-1} w_j \right) \frac{p_s}{w_s} \right\rfloor \tag{4}$$

and the corresponding *Dantzig integer solution* is $x_j = 1$ for $j < s$ and $x_j = 0$ for $j \geq s$.

No better bounds were presented in the following two decades until Martello and Toth [6] presented a tighter bound by imposing integrality on the *critical variable.* Since then, several other bounds have been presented based on Lagrangian relaxation, partial enumeration, construction of valid additional constraints, and different relaxations of the latter.

The first branch-and-bound algorithms for KP appeared in the early 1970s. Among the most successful we should mention the algorithms by: Horowitz and Sahni [5], Nauss [11] and Martello and Toth [6].

These algorithms are all based on a depth-first enumeration, in order to limit the space consumption. The branching strategy consists of selecting an item j and generating two children nodes through conditions $x_j = 1$ and $x_j = 0$. The items are examined in the order given by (2), and the branching node is selected as the (unique) active node generated by a condition $x_k = 1$ or, if there is no such node, as the last active node generated by a condition $x_k = 0$. Upper bounds for these algorithms are derived from some kind of continuous relaxation of the currently induced subproblem.

A comparison of the algorithms, presented in Martello and Toth [8], shows that they perform well for small sized "easy" instances.

3 Core algorithms

In order to solve large-sized instances, Balas and Zemel [1] proposed "guessing" the optimal values of several decision variables, and focus the enumeration on the most interesting ones. This subset of items, known as the *core* of the problem, is then solved either by heuristic techniques or by one of the above exact branch-and-bound methods. The core C could be determined by finding the critical variable, x_s, and setting $C = \{s - \delta, \ldots, s, \ldots, s + \delta\}$ in the sequence given by (2), for a prefixed value δ. In order to avoid explicit sorting, Balas and Zemel give a procedure based on partitioning techniques that determines both x_s and the core in $O(n)$ time.

All variables preceding the core are fixed to 1, while all those following it are fixed to 0 (i.e., for each item j not in the core, x_j is set to one, if $p_j/w_j \geq p_s/w_s$, or to 0 otherwise). The solution of the core problem yields a lower bound which in many cases corresponds to the optimal solution value. In order to prove optimality of the solution found, either an upper bound having the same value is determined, or reduction procedures are used to prove that items not in the core should have the above specific value. When optimality cannot be proved, items which were not fixed to a specific value by reduction are finally enumerated to optimality.

Effective algorithms based on a core problem were presented by Balas and Zemel [1], Fayard and Plateau [4] and Martello and Toth [7]. A computational comparison of core algorithms presented in [8] shows that the `mt2` algorithm by Martello and Toth gives the best performance.

A final refinement of the core approach was presented in Pisinger [12], where an *expanding core* was used to ensure that the core size automatically adapts to the hardness of the problem. The algorithm starts from the Dantzig integer solution, and at each iteration it either inserts a new item or removes an item, depending on whether the weight sum $\overline{w}$ of the chosen items is less than c or greater than c. Initially the core C contains only item s, but whenever a new item $j \notin C$ is considered, the core is automatically expanded by sorting some more items. Simple bounding rules obtained from linear relaxation were used to limit the search. With a core $C = \{a, \ldots, b\}$ an upper bound is found to be

$$u(\overline{p}, \overline{w}) = \begin{cases} \overline{p} + (c - \overline{w})p_{b+1}/w_{b+1} & \text{if } \overline{w} \leq c, \\ \overline{p} + (c - \overline{w})p_{a-1}/w_{a-1} & \text{if } \overline{w} > c, \end{cases} \tag{5}$$

when the currently chosen items have profit sum $\overline{p}$ and weight sum $\overline{w}$. The algorithm backtracks whenever $\lfloor u(\overline{p}, \overline{w}) \rfloor \leq z$, where z is the incumbent solution value.

The best core-algorithms can solve "easy" instances with more than 100,000 items in less than a second. The "hard" instances can, however, only be solved for tiny instances.

4 Dynamic programming algorithms

Dynamic programming based on the Bellman recursion [2] is generally not an effective way of solving KP, since the space consumption is very large, and the worst-case and best-case computational efforts are generally the same.

Pisinger [13] presented a dynamic programming recursion, `minknap`, which starts from the Dantzig integer solution and at each iteration either inserts or removes an item. Thus assuming that $f_{a,b}(\tilde{c})$ for $a \leq s,\ b \geq s-1,\ 0 \leq \tilde{c} \leq 2c$ is an optimal solution to the core problem:

$$f_{a,b}(\tilde{c}) = \max \left\{ \begin{array}{l} \sum_{j=1}^{a-1} p_j + \sum_{j=a}^{b} p_j x_j : \\ \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j x_j \leq \tilde{c}, \\ x_j \in \{0,1\} \text{ for } j = a, \ldots, b \end{array} \right\}, \tag{6}$$

we may use the following recursion for the enumeration

$$f_{a,b}(\tilde{c}) = \max \begin{cases} f_{a,b-1}(\tilde{c}) & \text{if } b \geq s,\ \tilde{c} \geq 0 \\ f_{a,b-1}(\tilde{c} - w_b) + p_b & \text{if } b \geq s,\ \tilde{c} - w_b \geq 0 \\ f_{a+1,b}(\tilde{c}) & \text{if } a < s,\ \tilde{c} \leq 2c \\ f_{a+1,b}(\tilde{c} + w_a) - p_a & \text{if } a < s,\ \tilde{c} + w_a \leq 2c \end{cases} \tag{7}$$

Assuming that $\hat{p},\ \hat{w}$ are the profit and weight sums of the Dantzig integer solution, we may initially set $f_{s,s-1}(\tilde{c}) = -\infty$ for $\tilde{c} = 0, \ldots, \hat{w} - 1$ and $f_{s,s-1}(\tilde{c}) = \hat{p}$ for $\tilde{c} = \hat{w}, \ldots, 2c$. An optimal solution to KP is then given by $f_{1,n}(c)$ by alternatively increasing b and decreasing a. Upper bounds similar to (5) are used to fathom states in the recursion, and the enumeration is terminated as soon as optimality of the current incumbent solution can be proved. This implies that a minimal core is enumerated, and although the worst-case time complexity is $O(nc)$ as for the Bellman recursion, most instances can be solved without enumerating too many variables. In addition, since forward recursions are used in dynamic programming, very few states are generated.

5 Tighter bounds

Martello and Toth [9] presented an effective branch-and-bound algorithm, `mth`, where additional constraints on the minimum and maximum cardinality of an optimal solution are generated from extended covers.

The idea is to determine a value K such that

$$\sum_{j=1}^{n} x_j \leq K \tag{8}$$

in any optimal solution (x) to KP. We can then obtain an equivalent ILP model, $\mathrm{KP}^{\leq}$, by adding (8) to (1). Whenever the continuous solution to KP violates (8) (i.e., the critical variable is $s > K$, with $x_s > 0$), a better upper bound is given by the solution value of the continuous relaxation of $\mathrm{KP}^{\leq}$. This solution is efficiently determined, without using an LP solver, by Lagrangian relaxing (8) into the objective function. Using a multiplier $\lambda \geq 0$ and relaxing the integrality constraints, one gets the problem given by

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j - \lambda \left(\sum_{j=1}^{n} x_j - K \right) = \sum_{j=1}^{n} \tilde{p} x_j + \lambda K \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & 0 \leq x_j \leq 1, \qquad j = 1, \ldots, n, \end{aligned} \tag{9}$$

which is a continuous KP. The optimal multipliers λ may be derived using a specialized binary search method, since Martello and Toth prove some monotonicity properties of the above problem.

When the continuous solution to KP does not violate (8) a minimum cardinality constraint $\sum_{j=1}^{n} x_j \geq k$ is imposed on the problem, and the relaxation is solved in a similar way.

The addition of cardinality constraints seems to be an efficient technique for closing the gap between the LP and ILP optimum, but the bounds are relatively expensive to derive. In order to obtain good performance also for "easy" instances, the `mth` algorithm first tries to solve the problem by means of the simpler `mt2` approach, and only if this does not succeed within a given time limit, the more expensive bounds are derived. Dynamic programming applied to the last items was also used in order to save time in the enumeration, when searching for some items to fill the residual capacity.

6 A combined approach

Martello, Pisinger and Toth [10] proposed to combine dynamic programming with tight bounds, obtaining an algorithm, `combo`, with time complexity $O(nc)$. In general the worst-case bound is very pessimistic since most instances are solved very quickly due to the tight bounds. Upper bounds are derived by imposing cardinality constraints of the form (8), but these are surrogate relaxed with the original capacity constraint leading to a new 0-1 Knapsack Problem. Using a multiplier $\sigma \geq 0$ one gets the problem

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} (w_j + \sigma) x_j \leq c + \sigma K \\ & x_j \in \{0,1\}, \quad j = 1, \ldots, n. \end{aligned} \tag{10}$$

A good choice of multiplier σ is found by means of binary search in a similar way as in Martello and Toth [9]. The new problem (10) tends to be much easier to solve, since continuous bounds for this problem are generally tight. An optimal solution to the transformed problem yields an upper bound on the original problem, but if the cardinality of the solution found is correct, i.e. if $\sum_{j=1}^{n} x_j \leq K$ in the optimal solution to (10), one also obtains a feasible solution to the original problem, thus solving the problem to optimality.

In order to solve some special non-filling knapsack instances, rudimentary divisibility arguments are used to decrease the capacity whenever possible. Let d be the greatest common divisor of the weights $w_1, \ldots, w_n$. If $d \neq 1$, the capacity may be decreased to $c = d\lfloor c/d \rfloor$. Deriving d can be done in $O(n \log \max\{w_j\})$ time using Euclid's algorithm.

The enumeration part of `combo` is based on the dynamic programming recursion (7), but initially a *core* is chosen as a collection of items which fit together well with respect to some heuristic algorithms. Moreover, when the number of states in the dynamic programming gets too high, the lower bound is improved by pairing states with items not in the core. This usually results in a tightening of the lower bound and thus in additional fathoming of states.

7 Computational experiments

We have investigated how the most effective algorithms behave for different instance types and data ranges. Seven types of randomly generated data instances are considered as sketched below (with, in brackets, the abbreviations used in the tables). Each type is tested with *data range* $R = 1000$ and $10\,000$ for different problem sizes n.

- *Uncorrelated instances* (unc.): the weights w_j and the profits p_j are uniformly random distributed in $[1, R]$.
- *Weakly correlated instances* (weakly cor.): the weights w_j are distributed in $[1, R]$, and the profits p_j in $[w_j - R/10, w_j + R/10]$ such that $p_j \geq 1$.
- *Strongly correlated instances* (str.cor.): the weights w_j are distributed in $[1, R]$, and the profits are set to $p_j = w_j + R/10$.

- *Inverse strongly correlated instances* (inv.str.cor.): the profits p_j are distributed in $[1, R]$, and the weights are set to $w_j = p_j + R/10$.
- *Almost strongly correlated instances* (alm.str.cor.): the weights w_j are distributed in $[1, R]$, and the profits p_j in $[w_j + R/10 - R/500, w_j + R/10 + R/500]$.
- *Subset-sum instances* (subset-sum): the weights w_j are randomly distributed in $[1, R]$ and the profits are set to $p_j = w_j$.
- *Uncorrelated instances with similar weights* (unc.sim.w.): the weights w_j are distributed in $[100000, 100100]$ and the profits p_j in $[1, 1000]$.

For instance h in a series of $H = 100$ instances, the capacity is chosen as $c = \frac{h}{H+1} \sum_{j=1}^{n} w_j$. All tests were run on a HP9000-735/99, and a time limit of 5 hours was put on each series of 100 instances.

Tables 1 to 4 compare the average solution times of `mt2`, `minknap`, `mth` and `combo`. The oldest of the codes, `mt2`, is not able to solve the "hard" instances, but it performs well for uncorrelated, weakly correlated and subset-sum instances. The `minknap` algorithm has overall stable behavior due to the pseudo-polynomial time bound, but the hard problems demand tens or hundreds of seconds to be solved. On the other hand, the solution times have a very stable growth for increasing data ranges R and problem sizes n. The `mth` algorithm can in most cases solve the instances faster than `minknap` due to the tighter upper bounds. There are, however, a few anomalous occurrences for the almost strongly correlated instances where the cardinality bounds do not work efficiently, and thus a very large enumeration takes place. The combined approach in `combo` is however clearly superior to all the previous approaches, being able to solve all the instances in less than 0.2 seconds. The solution times are very stable, and in nearly all instances `combo` is faster than all the previous approaches.

Table 1: Average cpu times for `mt2`, in HP9000-735/99 seconds

	unc.		weakly cor.		str.cor.		inv.str.cor.		alm.str.cor.		subset-sum		unc.sim.w.
$n \setminus R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.00	0.00	0.00	0.00	0.06	0.04	0.01	0.02	0.03	0.03	0.00	0.01	0.02
100	0.00	0.00	0.00	0.00	26.26	24.78	4.44	—	5.90	16.02	0.00	0.01	3.28
200	0.00	0.00	0.00	0.00	—	—	—	—	—	—	0.00	0.02	—
500	0.00	0.00	0.01	0.01	—	—	—	—	—	—	0.00	0.02	—
1000	0.00	0.01	0.01	0.02	—	—	—	—	—	—	0.00	0.02	—
2000	0.01	0.01	0.01	0.04	—	—	—	—	—	—	0.00	0.02	—
5000	0.01	0.02	0.01	0.08	—	—	—	—	—	—	0.01	0.02	—
10000	0.02	0.05	0.02	0.13	—	—	—	—	—	—	0.01	0.03	—

Table 2: Average cpu times for `minknap`, in HP9000-735/99 seconds

	unc.		weakly cor.		str.cor.		inv.str.cor.		alm.str.cor.		subset-sum		unc.sim.w.
$n \setminus R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.00	0.01	0.00	0.03	0.00
100	0.00	0.00	0.00	0.00	0.02	0.17	0.01	0.18	0.01	0.03	0.00	0.03	0.00
200	0.00	0.00	0.00	0.00	0.05	0.82	0.04	0.65	0.04	0.15	0.00	0.03	0.01
500	0.00	0.00	0.00	0.00	0.20	2.52	0.19	2.80	0.16	0.88	0.00	0.03	0.03
1000	0.00	0.00	0.00	0.01	0.48	8.30	0.45	7.59	0.37	3.18	0.00	0.03	0.10
2000	0.00	0.00	0.00	0.01	0.96	13.17	1.09	14.16	0.72	8.57	0.00	0.03	0.35
5000	0.00	0.01	0.01	0.02	3.73	54.11	3.20	54.66	1.63	26.57	0.01	0.04	1.32
10000	0.01	0.01	0.01	0.03	8.18	115.41	6.57	122.84	1.83	48.33	0.01	0.04	1.57

Table 3: Average cpu times for `mth`, in HP9000-735/99 seconds

	unc.		weakly cor.		str.cor.		inv.str.cor.		alm.str.cor.		subset-sum		unc.sim.w.
$n \setminus R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.03	0.00	0.01	0.00
100	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.01	0.03	0.14	0.00	0.01	0.01
200	0.00	0.00	0.00	0.00	0.04	0.05	0.03	0.04	0.06	0.36	0.00	0.02	0.03
500	0.00	0.00	0.01	0.01	0.09	0.09	0.08	0.09	0.10	0.75	0.00	0.01	0.06
1000	0.01	0.01	0.01	0.02	0.15	0.23	0.14	0.16	0.19	1.01	0.00	0.02	0.11
2000	0.01	0.02	0.01	0.03	0.17	0.38	0.18	0.23	0.31	0.81	0.00	0.01	0.18
5000	0.02	0.04	0.02	0.06	0.17	1.75	0.33	0.66	2.55	1.46	0.00	0.02	0.24
10000	0.04	0.08	0.02	0.10	0.28	5.89	0.48	1.64	—	—	0.01	0.68	0.35

Table 4: Average cpu times for `combo`, in HP9000-735/99 seconds

	unc.		weakly cor.		str.cor.		inv.str.cor.		alm.str.cor.		subset-sum		unc.sim.w.
$n \setminus R$	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^3	10^4	10^5
50	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.03	0.00
100	0.00	0.00	0.00	0.00	0.01	0.03	0.01	0.03	0.01	0.02	0.00	0.02	0.00
200	0.00	0.00	0.00	0.00	0.02	0.04	0.02	0.04	0.02	0.04	0.00	0.03	0.01
500	0.00	0.00	0.00	0.00	0.02	0.05	0.02	0.04	0.02	0.02	0.00	0.03	0.03
1000	0.00	0.00	0.01	0.01	0.02	0.07	0.03	0.06	0.03	0.02	0.00	0.03	0.06
2000	0.00	0.00	0.00	0.01	0.03	0.05	0.04	0.06	0.03	0.03	0.00	0.03	0.07
5000	0.01	0.01	0.01	0.03	0.04	0.05	0.04	0.06	0.04	0.04	0.00	0.03	0.12
10000	0.01	0.02	0.01	0.04	0.08	0.07	0.08	0.09	0.07	0.08	0.01	0.02	0.12

8 Conclusions

The current decade has considerably extended the classes of knapsack problems which can be effectively solved. New tight bounds have been derived from cardinality constraints, and better dynamic programming recursions have been developed, which makes it possible to terminate the enumeration before all the items have been considered. Combining tight bounds and dynamic programming leads to a good trade-off between worst-case and best-case behavior. Future research will show whether this development may be continued so as to be able to solve all instances with bounded weights in a reasonable amount of time.

Acknowledgements

The first and third author acknowledge Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) and Consiglio Nazionale delle Ricerche (CNR) for the support of this project. The second author would like to thank the EC Network DIMANET for supporting the research by European Research Fellowship No. ERBCHRXCT-94 0429.

References

[1] E. Balas and E. Zemel (1980), "An algorithm for large zero-one knapsack problems", *Operations Research*, **28**, 1130–1154.

[2] R. E. Bellman (1957), *Dynamic programming*, Princeton University Press, Princeton, NJ.

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

[4] D. Fayard and G. Plateau (1982), "An Algorithm for the solution of the 0-1 knapsack problem", *Computing*, **28**, 269–287.

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

[6] S. Martello and P. Toth (1977), "An upper bound for the zero-one knapsack problem and a branch and bound algorithm", *European Journal of Operational Research*, **1**, 169–175.

[7] S. Martello and P. Toth (1988), "A new algorithm for the 0-1 knapsack problem", *Management Science*, **34**, 633–644.

[8] S. Martello and P. Toth (1990), *Knapsack problems: Algorithms and computer implementations,* Wiley, Chichester, England.

[9] S. Martello and P. Toth (1993), “Upper bounds and algorithms for hard 0-1 knapsack problems”, *Research Report DEIS, University of Bologna*, OR/93/04. To appear in *Operations Research.*

[10] S. Martello, D. Pisinger and P. Toth (1997), “Dynamic programming and tight bounds for the 0-1 knapsack problem”, *Research Report DEIS, University of Bologna*, OR/97/1.

[11] R. M. Nauss (1976), “An efficient algorithm for the 0-1 knapsack problem”, *Management Science*, **23**, 27–31.

[12] D. Pisinger (1995), “An expanding-core algorithm for the exact 0-1 knapsack problem,” *European Journal of Operational Research*, **87**, 175–187.

[13] D. Pisinger (1994), “A minimal algorithm for the 0-1 knapsack problem”, *DIKU, University of Copenhagen, Denmark,* Report 94/23. To appear in *Operations Research.*