# Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem

**Silvano Martello[1], David Pisinger[2], Paolo Toth[1]**

[1]DEIS, University of Bologna, Viale Risorgimento 2, Bologna

[2]DIKU, University of Copenhagen, Univ.parken 1, Copenhagen

**Abstract**

Two new algorithms recently proved to outperform all previous methods for the exact solution of the 0-1 Knapsack Problem. This paper presents a combination of such approaches, in which, in addition, valid inequalities are generated and surrogate relaxed, and a new initial core problem is adopted. The algorithm is able to solve all classical test instances, with up to 10,000 variables, within less than 0.2 seconds on a HP9000-735/99 computer. The C language implementation of the algorithm is available on the internet.

## 1 Introduction

We consider the classical *0-1 Knapsack Problem* (KP) where a subset of $n$ items has to be packed into a knapsack of capacity $c$. Each item $j$ has a profit $p_j$ and a weight $w_j$, and the problem is to maximize the profit sum of the chosen items without exceeding the capacity $c$. Thus we have the integer linear programming (ILP) model:

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

where $x_j$ takes the value 1 iff item $j$ is packed. Without loss of generality, all coefficients $p_j, w_j$ and $c$ are assumed to be positive integers. To avoid trivial cases we assume that $w_j \le c$ for all $j = 1,\dots,n$ and that $\sum_{j=1}^{n} w_j > c$

This NP-hard problem has important managerial applications, especially in the cutting and packing area and in loading contexts, and it arises frequently as a subproblem in the solution of more complex optimization problems. As a consequence, it has aroused great interest during the last two decades, and several effective algorithms have emerged for its solution (see Martello and Toth [6] for a comprehensive survey).

As a state of the art, we can mention the following. Instances where a loose correlation, or no correlation at all, exists among the profit and weight of each item (*weakly correlated* and *uncorrelated* instances) can easily be solved to optimality even for large values of $n$, while *strongly correlated* instances, as well as instances involving very large profit and weight values, may be very difficult. Recently much effort has been made to solve the latter category of instances. Martello and Toth [7] presented a new branch-and-bound scheme, where additional cardinality constraints are generated from extended covers and relaxed in a Lagrangian way. This approach seems to be efficient to close the gap between the LP and ILP optimum, but the bounds are relatively expensive to derive. Pisinger [11] on the other hand used simple LP bounds in the dynamic programming enumeration, but, due to a new way for initializing and expanding the core problem, he was able to limit the enumeration considerably. The same author [10] also proposed a new approach for closing the gap between the LP and ILP, which is extremely efficient for strongly correlated instances, but cannot solve general knapsack problems.

In this paper we investigate a combination of the above approaches, and introduce new upper bounds obtained from the surrogate relaxation of cardinality constraints, as well as a new way for generating an initial core. Other hybrid algorithms for knapsack and subset-sum problems have been presented by Martello and Toth [4], and Plateau and Elkihel [14].

A general framework of the new algorithm is given in Section 2, while the detailed description of its components is presented in Sections 3–5. Extensive computational experiments are presented and discussed in Section 6. The C language implementation of the algorithm is available on the internet at `http://www.diku.dk/~pisinger/codes.html`.

In the following, given an array $q$ ($q \in \{p, w, x\}$), we sometimes use, for the sake of brevity, the notation $Q(k) = \sum_{j=1}^{k} q_j$.

# 2 General outline of the algorithm

The LP relaxation of (1) can be solved by the *greedy algorithm*: assume that the items are ordered by non-increasing profit-to-weight ratios $p_j/w_j$, and fill the knapsack until the first item which does not fit is found. The decision variable corresponding to this *break item* $b$,

$$b = \min\{h : W(h) > c\} \tag{2}$$

is set to $x_b = (c - W(b-1))/w_b$, while all items before $b$ are chosen ($x_j = 1$, for $j = 1, \ldots, b-1$) and no item after $b$ is chosen ($x_j = 0$ for $j = b+1, \ldots, n$). The corresponding objective value, rounded down to the closest integer, is known as the *Dantzig upper bound*

$U_D = P(b-1) + \lfloor (c - W(b-1))p_b/w_b \rfloor$, while $P(b-1)$ is the value of a feasible solution, known as the *break solution*.

Balas and Zemel [1] presented an effective algorithm for large-sized easy knapsack problems, which determines the break item (and hence the Dantzig solution) in $O(n)$ time, and restricts the enumeration to a small subset $C$ of items having profit-to-weight ratios close to that of the break item (the *core* problem). Our initial core, instead, contains items whose profit-to-weight ratio is not related, in general, to that of the break item. The core is enumerated through dynamic programming, introducing new items when needed.

The basic recursion (for which the reader is referred to algorithm `minknap` in Pisinger [11]) has $O(nc)$ time complexity and generates, at most, $2c$ undominated states. It is well suited for solving easy problems, so its main structure is unchanged until difficult instances are met. Chvátal [2] has considered a family of algorithms classified according to the use of dominance relations, rudimentary divisibility properties, or bounding rules. Difficult instances can be constructed depending on how many of the three techniques are used in an algorithm. We use all three techniques in order to obtain a very tight model and characterize difficult instances by the fact that the number of dynamic programming states grows beyond given threshold values. Our overall approach is outlined below. Steps 1, 3 and 4 are discussed in detail in the following sections.

1. We start the dynamic programming recursion with a core different from the one used in [11] (see Section 3), which is likely to produce a well-filled knapsack.

2. If the number of states grows beyond a given value $M_1$, we derive the greatest common divisor $d$ of the weights: if $d \neq 1$ we decrease the capacity to $c = d\lfloor c/d \rfloor$. We compute $d$ in $O(n \log \max\{w_j\})$ time through Euclid's algorithm.

3. If the number of states exceeds a given value $M_2 > M_1$, we derive a minimum or maximum cardinality constraint, surrogate relax it with the original weight constraint, and solve the relaxed problem to optimality by means of the same knapsack algorithm. This yields a good upper bound and, in many cases, even an optimal solution to the original problem (see Section 4). This approach is interesting, since we transform a difficult problem into an easier one which has the same structure but can be solved in reasonable time. The solution of the transformed problem yields enough information about the original problem to considerably speed up the solution process.

4. If the number of states exceeds a given value $M_3 > M_2$ we try to improve the current lower bound by pairing dynamic programming states with items outside the core

(see Section 5). This frequently gives a lower bound equal to the upper bound.

5. If the previous attempts fail, we have good upper and lower bounds. The algorithm continues as the `minknap` algorithm ([11]), but with upper and lower bounds better than those of `minknap`, thus speeding up the solution process.

The values of $M_1, M_2$ and $M_3$ have experimentally been set to 1000, 2000 and 10000, respectively. However these values are not critical: easy instances (or instances which become easy after scaling) are usually solved far before the $M_1$ (or $M_2$) limit is reached, while for difficult instances the number of preliminar iterations is inessential.

The break item $b$ is found through partial sorting in $O(n)$ time, using the technique in [11]. The algorithm also returns some partially sorted intervals which can be used later for expanding the core without complete sorting of the items.

# 3 The initial core

Recently Pisinger [13] showed that solving the classical core problem (Balas and Zemel [1], Martello and Toth [5,6]) can require, in some specific situations, very high solution times due to degeneration: if the item weights in the core are close to each other, it is very difficult to obtain a well filled knapsack. For the strongly correlated instances a core of 450 items is often necessary to find an optimal solution, while uncorrelated and weakly correlated instances can be solved with a core of size 30. Difficult knapsack instances are frequently characterized by optimal solutions having a number of chosen items very close to $b$. For example, as observed by Pisinger [10], an optimal solution to any strongly correlated instance includes exactly $b-1$ items. Hence we construct an initial core by first selecting items $\{b, b-1\}$, and adding items which fit well together, allowing a good knapsack filling with a number of items close to $b$.

The first two additional items, $\gamma$ and $\gamma'$, are found by using the forward and backward greedy algorithms from [9]: $\gamma$ is the item of highest profit which can be added to the break solution once item $b-1$ has been removed, while $\gamma'$ is the item of lowest profit which has to be removed from the break solution so that item $b$ can be added.

In order to handle other difficult instances, where an optimal solution consists of $b$ items, two more items, $\beta$ and $\beta'$, obtained through variants of the above heuristics, are inserted into the core: $\beta$ is the item of highest profit which can be added to the break solution, while $\beta'$ is the item of lowest profit which has to be removed from the break solution so that items $b$ and $b+1$ can be added.

We complete the initial core with the two items $\rho$ and $\rho'$ having the smallest and largest weight, in order to ensure some variation in the weights. The eight or less items of the resulting core $C$ (note that some of the additional items could not exist) are sorted according to nonincreasing profit-to-weight ratios, and the dynamic programming enumeration is performed on $C$, by assuming that $x_j = 1$ (resp. $x_j = 0$) for each item $j \notin C$ that precedes (resp. follows) $b$ in the profit-to-weight ordering. The initial lower bound $z$ is set to the highest profit of a feasible state, and the upper bound $U$ to the Dantzig bound $U_D$.

A relevant difference between our core and that used by Pisinger [11] is that the latter is initialized to $\{b, b-1\}$ and consists of consecutive items $[s, s+1, \ldots, b, \ldots, t-1, t]$ (with respect to the profit-to-weight ordering), while $C$ generally contains non-consecutive items. The original dynamic programming recursion is modified accordingly.

# 4 Bounds from cardinality

When the LP solution is far from the ILP solution it may be useful to add some additional constraints derived from considerations on the minimum or maximum cardinality of an optimal solution. Such constraints do not exclude any integer solution, but they tighten the LP model so that the LP solution comes closer to the ILP solution. Cardinality bounds as presented by Martello and Toth [7] are derived as follows: assume the items are ordered by non-decreasing weight, and let $k = \min\{h : W(h) > c\} - 1$. Then a *maximum cardinality constraint* of the form

$$\sum_{j=1}^{n} x_j \leq k \tag{3}$$

can be added to problem (1) without excluding any feasible solution.

However, adding a constraint which is not violated in the LP solution does not tighten the formulation, thus we add a maximum cardinality constraint only if $k = b - 1$. In addition, if an optimal solution to the problem given by (1) with the capacity constraint replaced by (3) (easily obtained by taking the $k$ items of highest profit, breaking ties by lowest weight) satisfies the capacity constraint, then this solution is also optimal for (1).

In a similar way we can define a *minimum cardinality constraint* (see [7]). Assume that the current lower bound is given by $z$ and that the items are ordered according to non-increasing profit. We set $k = \max\{h : P(h) \leq z\} + 1$, and thus have the constraint

$$\sum_{j=1}^{n} x_j \geq k \tag{4}$$

for any solution with objective value larger than $z$. As before, there is no reason to add this constraint to (1) if it is not violated, thus we will only use it when $k = b$. In addition, if no feasible solution to (1) satisfies (4) (i.e., if the sum of the $k$ smallest weights exceeds $c$) then we know that the solution producing $z$ is optimal.

Adding constraint (3) or (4) to our model leads to a two-constraint knapsack problem which may be difficult to solve. Thus we surrogate relax the cardinality constraint with the original weight constraint using surrogate multiplier values $S$ and 1, respectively. For the maximum (resp. minimum) cardinality constraint we use $S \geq 0$ (resp. $S \leq 0$). In both cases the relaxation leads to the KP (1) with capacity constraint replaced by

$$\text{subject to} \quad \sum_{j=1}^{n}(w_j + S)x_j \leq c + Sk \tag{5}$$

Let SKP denote the resulting problem, and LSKP its LP relaxation. A negative multiplier value $S$ can result in a non-positive weight. However, for each item $j$ of the relaxed problem with non-positive weight, we can set $x_j = 1$ and increase the capacity by $|w_j|$.

We are of course interested in the surrogate multiplier value which leads to the best upper bound for (1). But deriving it for SKP is difficult, since the objective values may change radically for small changes of $S$ and each tentative multiplier value requires the solution of an ordinary KP. In the next section we show that, for the LP relaxation LSKP, an optimal surrogate multiplier value can be found in $O(n^2)$ time, due to the monotonicity of the left-hand side of the cardinality constraint as a function of $S$.

## 4.1 Binary search for the multiplier value

We will show that an optimal multiplier value of LSKP can be derived through binary search in a way similar to that proposed by Martello and Toth [7] for the Lagrangian relaxation of the cardinality constraints. We describe the resulting methodology for the case of maximum cardinality. A similar approach can be easily derived for the case of minimum cardinality. Let $x(S)$ be an optimal solution to LSKP for a given multiplier value $S \geq 0$, and let $\Gamma(S) = \sum_{j=1}^{n} x_j(S)$. Assume that the items have been sorted according to non-increasing $p_j/(w_j + S)$ values, breaking ties by decreasing $w_j$ values, so that $\Gamma(S)$ is uniquely defined at breakpoints. We show that $\Gamma(S)$ is a monotonous function.

**Theorem 1** *$\Gamma(S)$ is monotonically non-increasing when $S$ increases.*

**Proof.** We will show that if $S$ increases and two items change place in the profit-to-weight ordering, then the item with larger weight moves to the front. Consider two items $r$ and $s$, and two values $S'$, $S''$ (with $S' < S''$) such that $p_r/(w_r + S') \geq p_s/(w_s + S')$

and $p_r/(w_r + S'') \leq p_s/(w_s + S'')$. By subtracting $(p_r(w_s + S') \geq p_s(w_r + S'))$ from $(p_r(w_s + S'') \leq p_s(w_r + S''))$ we get $p_r(S'' - S') \leq p_s(S'' - S')$, hence $p_r \leq p_s$. By observing that an item $s$ having a higher profit and a lower weight than an item $r$ will always precede $r$ in the profit-to-weight sorting (for any non-negative value of $S$), we also see that $w_r \leq w_s$, since otherwise $r$ and $s$ would violate the given conditions. It follows that $r$ and $s$ satisfy $p_r \leq p_s$ and $w_r \leq w_s$, from which we can conclude that, as $S$ increases, in the sequence sorted according to non-increasing $p_j/(w_j + S)$ ratios, the items with larger weights move to the first positions. Hence $\Gamma(S)$ never increases with $S$. □

Thus an optimal multiplier value $S^*$ (i.e., from complementary slackness, the one for which $\Gamma(S^*) = k$) can be determined through binary search: at each iteration the current value of $S$ is increased if $\Gamma(S) > k$, or decreased if $\Gamma(S) < k$. The procedure can be accelerated as follows. Given the current $S$ value, assume that the items are sorted according to non-increasing $p_j/(w_j + S)$ ratios and let $b(S)$ be the corresponding break item. We first consider the case $\Gamma(S) > k$: a higher value $S'$ giving the same solution as $S$ (i.e., requiring no search for a new break item), is given by the maximum value for which: (i) no item currently preceding $b(S)$ moves to the right of $b(S)$; (ii) no item currently following $b(S)$ moves to the left of $b(S)$. From the first part of the proof of Theorem 1 we know that condition (i) (resp. (ii)) holds if $p_j \geq p_{b(S)}$ (resp. $p_j \leq p_{b(S)}$). Thus

$$S' = \min \left\{ \frac{p_j w_{b(S)} - p_{b(S)} w_j}{p_{b(S)} - p_j} : (j < b(S) \text{ and } p_j < p_{b(S)}) \text{ or } (j > b(S) \text{ and } p_j > p_{b(S)}) \right\} \tag{6}$$

If instead we obtain $\Gamma(S) < k$, a similar argument leads to a smaller $S'$ producing the same solution as $S$. The result is given by (6) with "min" replaced by "max", and conditions "$j < b(S)$", "$j > b(S)$" interchanged.

**Theorem 2** *Let $U(S^*)$ be the value of the final solution $x(S^*)$. Then $x(S^*)$ gives an optimal solution to the LP relaxation of (1), (3), and the resulting upper bound $\lfloor U(S^*) \rfloor$ dominates the Dantzig upper bound $U_D$.*

**Proof.** The solution $x(S^*)$ satisfies $\sum_{j=1}^{n}(w_j + S^*)x_j(S^*) \leq c + S^*k$. Note that $\Gamma(S^*) = k$. Hence $\sum_{j=1}^{n} w_j x_j(S^*) \leq c$ and $\sum_{j=1}^{n} x_j(S^*) \leq k$. It follows that $x(S^*)$ satisfies the relaxed constraints, hence it is optimal for the LP relaxation of (1), (3). In addition, since $x(S^*)$ is feasible for the LP relaxation of (1), the corresponding upper bound dominates $U_D$.□

¿From (6) we know that the number of distinct multiplier values to be considered is bounded by $n(n-1)$: each resulting problem is solvable in $O(n)$ time, so a straightforward computation of $U(S^*)$ can be done in $O(n^3)$ time. A theoretically faster method could perform the binary search over the $n(n-1)$ possible multiplier values, by determining, at

each iteration, the median of the current subset: it is easily seen that the resulting time complexity would be $O(n^2)$. We have used a different polynomial algorithm, for which the computational experiments showed a much better average performance, although its complexity is bounded in the logarithm of the coefficients.

There are at most $n(n-1)$ distinct multiplier values $S$, each defined by a pair of items $(i,j)$ through $S = (p_j w_i - p_i w_j)/(p_i - p_j)$. Computational experiments however showed that the objective value of LSKP as a function of $S$ has a wide interval of multiplier values for which $U(S^*)$ is obtained, thus in practice only integer values of $S$ need be considered. It is easily seen that the largest value of $S$ we need to consider is $S_{\max} = p_{\max} w_{\max} - 1$ where $p_{\max} = \max_{j=1,\dots,n} p_j$ and $w_{\max} = \max_{j=1,\dots,n} w_j$.

Thus, starting with $S_1 = 0$ and $S_2 = S_{\max}$, we repeatedly derive $S' = \lfloor (S_1 + S_2)/2 \rfloor$: if $\Gamma(S') > k$ we set $S_1 = S'$, and if $\Gamma(S') < k$ we set $S_2 = S'$. This binary search is continued until $S_1 = S_2$ or $\Gamma(S') = k$. Since for each value of $S'$ the resulting problem can be solved in $O(n)$ time, we obtain an algorithm running in $O(n \log(S_{\max})) = O(n \log \max\{p_{\max}, w_{\max}\})$.

## 4.2 Improvement of the upper bound

In some difficult instances it is not possible to impose an additional constraint from minimum or maximum cardinality (i.e., we obtain $k \geq b$ for the maximum cardinality, and $k < b$ for the minimum cardinality). In these situations we force the cardinality by partitioning the feasible solution set into two subsets. This is obtained by generating two subproblems $P_1$ and $P_2$, given by the original problem (1) with the additional constraint $\sum_{j=1}^{n} x_j \leq b-1$ (resp. $\sum_{j=1}^{n} x_j \geq b$), and determining the corresponding upper bounds $U_1$, $U_2$. A valid upper bound for (1) is then $U = \max\{U_1, U_2\}$.

Having found an optimal $S^*$ value for LSKP, if $U(S^*)$ is greater than the incumbent solution value, we solve SKP with $S = S^*$. This is convenient for two reasons: (i) the bound given by the resulting integer solution $\overline{x}(S^*)$ may be better than the bound $U(S^*)$ given by the LP solution $x(S^*)$; (ii) $\overline{x}(S^*)$ may be feasible (hence optimal) for the original problem (1). Computational experiments have shown that cases (i) and (ii) occur frequently. As seen in Section 4.1, we consider integer multiplier values, thus obtaining an ordinary KP. Hence the resulting instance is solved through a special call to our overall algorithm (see Section 2): in this case we set $M_2 = +\infty$, so as to avoid loops.

# 5 Improvement of the lower bound

The upper bounds of the previous section are generally very good, and usually equal the optimal solution value. However, for highly degenerate problems, in which the items in the core $C$ have weights very close to each other, it may be difficult to obtain a good lower bound in acceptable time. Indeed for such instances, when the residual capacity of the break solution is large, it is very difficult to obtain a solution filling the knapsack.

In such situations an improved lower bound can be obtained by finding optimal pairs consisting of an item and one of the states generated by the dynamic programming recursion. Let $\pi_i$ and $\mu_i$ be the total profit and weight of state $i$. For each item $j \notin C$ that precedes $b$ in the profit-to-weight ordering, we find the state $i$ with largest $\mu_i$ (hence also largest $\pi_i$, since dominated states are removed) such that $\mu_i \leq c + w_j$: the corresponding objective value is $z_j = \pi_i - p_j$. Similarly, for each item $j \notin C$ that follows $b$, we find the state $i$ with the largest $\mu_i$ such that $\mu_i \leq c - w_j$: the corresponding objective value is $z_j = \pi_i + p_j$. The best $z_j$ ($j \notin C$) is then selected for the lower bound improvement.

Since the states are ordered by increasing weights (and profits), we can find the required $\mu_i$ through binary search, in $O(\log M_3)$ time, where (see Section 2) $M_3 \leq 2c$ is the number of states. This gives a total time of $O(n \log M_3)$ for the lower bound improvement.

# 6 Computational experiments

The algorithm described in the previous sections was coded in C language, and we refer to it as `combo`. We examine its behavior for different problem sizes, instance types, and data ranges. Nine types of randomly generated data instances from the literature were considered. Each type but the last one was tested with two data ranges: $R = 1000$, $R = 10000$ (smaller $R$ values produced very easy instances). The instance types are:

- *Uncorrelated*: $w_j$ and $p_j$ uniformly random in $[1, R]$.
- *Weakly correlated*: $w_j$ uniformly random in $[1, R]$, $p_j$ uniformly random in $[w_j - R/10, w_j + R/10]$ so that $p_j \geq 1$.
- *Strongly correlated*: $w_j$ uniformly random in $[1, R]$, $p_j = w_j + R/10$.
- *Inverse strongly correlated*: $p_j$ uniformly random in $[1, R]$, $w_j = p_j + R/10$.
- *Almost strongly correlated*: $w_j$ uniformly random in $[1, R]$, $p_j$ uniformly random in $[w_j + R/10 - R/500, w_j + R/10 + R/500]$.

- *Subset-sum*: $w_j$ uniformly random in $[1, R]$, $p_j = w_j$.
- *Even-odd subset-sum*: $w_j$ uniformly random in $[1, R]$ but even, $p_j = w_j$, $c$ odd.
- *Even-odd strongly correlated:* $w_j$ uniformly random in $[1, R]$ but even, $p_j = w_j + R/10$, $c$ odd.
- *Uncorrelated with similar weights:* $w_j$ uniformly random in $[10^5, 10^5 + 100]$, $p_j$ uniformly random in $[1, 1000]$.

For each instance type, data range and value of $n$ (ranging from 50 to 10000), we generated 100 instances, with $c = \frac{h}{101}W(n)$ for instance number $h$. In the even-odd instances, the capacity was however rounded up to the nearest odd number. 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 algorithms `mt2` (Martello and Toth [6]), `minknap` (Pisinger [11]), `mth` (Martello and Toth [7]) 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 an overall stable behavior due to the pseudo-polynomial time bound, but `mth` can in most cases solve the instances faster due to the better upper bounds. For the even-odd instances `minknap` is able to solve reasonably large problems where `mth` can only solve tiny instances. The combined approach in `combo` is however clearly superior to the previous approaches, being able to solve all the instances with average times smaller than 1/8 seconds.

## 6.1 A closer study of `combo`

Dynamic programming algorithms often demand much memory. Table 5 shows the maximum number of states needed by the `combo` algorithm: it never generates more than 140,000 states, using less than 2Mb of memory. There are three reasons for this nice behavior: i) the dynamic programming recursion starts in the middle of the instance (see Pisinger [11]), so it does not need enumerate all items; ii) a *forward* dynamic programming recursion is used, so a large amount of dominated states can be fathomed; iii) in many cases the bounds make it possible to terminate the enumeration with a small core. The average core size is given in Table 6. For most instances the enumerated core is less than 50 items, although some difficult problems may demand a core of size up to 500 items.

Finally Table 7 shows how many surrogate relaxations were solved in order to derive a better upper bound. With the chosen value of $M_2$ ($M_2 = 2000$), very few of the easy instances require the relaxed problem to be solved, which is desirable since the

continuous bound is sufficient for solving uncorrelated and weakly correlated instances. For the difficult instance types, nearly all the instances benefit from the better bounds. Additional experiments showed that the SKP solution is an optimal solution to the original KP in 80-95% of the (large sized) almost strongly correlated instances, while for the other instances only a couple of the relaxed solutions had the right cardinality.

## 6.2 Classes of hard problems

The `combo` algorithm solves all classes of instances from the KP literature, so we have tried to identify instance types which cannot be handled so easily. Chvátal [2] showed that very difficult instances can be constructed by using very large weights. However, in practical applications one always has a bound on the sizes, if not for other reasons, because of the word-length of an integer. Thus we tried instances which are difficult to solve even with weights in a limited range. If nothing else is stated, the capacity $c$ was chosen as previously:

- *Avis subset-sum (*see *[2]:* here we use weights of magnitude $O(n^2)$. We have $p_j = w_j = n(n+1)+j$, $c = n(n+1)\ \lfloor (n-1)/2 \rfloor\ + n(n-1)/2$. No subset of items meets the capacity constraint with equality, and no common divisor larger than one exists. The order of the items was randomly permuted to obtain 100 "different" instances.

- *Avis knapsack:* this is a generalization to knapsack: we have $p_j$ uniformly random in $[1, 1000]$, $w_j = n(n+1)+j$, $c = n(n+1)\ \lfloor (n-1)/2 \rfloor\ + n(n-1)/2$.

- *Collapsing knapsack:* Pferschy, Pisinger and Woeginger [8] showed that the collapsing knapsack problem can be transformed to an equivalent KP which, however, is difficult to solve, as it contains many additional constraints "hidden" in the weight inequality. We generate the instances as in Fayard and Plateau [3] with profits, weights and capacities uniformly random in $[1, 300]$, $[1, 1000]$ and $[1, 10000]$, respectively. Each instance is then transformed to a KP as shown in [8].

- *Bounded strongly correlated:* Pisinger and Toth [12] showed that when a bounded knapsack problem (where $x_j \in \{0, 1\}$ is replaced by $x_j \in \{0, 1, \ldots, m_j\}$) is transformed to a KP, the cardinality constraints loose their effect. Thus we generate bounded instances with $w_j$ uniformly random in $[1, 1000]$ and $p_j = w_j + 100$. The bounds $m_j$ are uniformly random in $[1, 10]$, and we transform the instance to a KP using the technique described in Martello and Toth [6], until $n$ items are present.

- *No small weight:* the unbounded knapsack problem becomes difficult when there are no small weights (see [12]): thus we generate $w_j$ uniformly random in $[500, 1000]$ and $p_j = w_j + \alpha$, with $\alpha$ uniformly random in $[-R/10, +R/10]$ such that $p_j \geq 1$.

The average solution times of 100 instances are given in Table 8. The Avis subset-sum instances are solved up to $n = 200$, while specialized algorithms can solve instances of size up to $n = 1000$ [12]: the performance of `combo` is not particularly bad, although the algorithm cannot benefit from the improvement of bounds nor from rudimentary divisibility. It is however interesting to see that the knapsack counterpart is easy, since several items and states can be fathomed by dominance and reduction. The collapsing knapsack instances could only be generated up to $n = 200$ on the present computer, and `combo` was able to solve them in reasonable time: the solution times are more or less the same as those reported in [8], indicating that the better bounds of `combo` do not accelerate the solution process. Also for the instances obtained by transforming bounded knapsack problems, the cardinality constraints lose their effect: `combo` solves the instances in 10 seconds while similar KP instances are solved 100 times faster, indicating that specialized algorithms for the bounded version should be developed in order to fully exploit the cardinality bounds. Finally, the instances with no small weight turn out to be exteremly easy for `combo`.

# Acknowledgements

We thank the EC Network DIMANET for supporting this research by Fellowship No. ERBCHRXCT-94 0429. We also thank MURST and CNR, Italy. We are grateful to two anonymous referees for helpful comments which considerably improved the presentation.

# References

[1] E. Balas and E. Zemel (1980), "An Algorithm for Large Zero-One Knapsack Problems", *Operations Research* 28, 1130–1154.

[2] V. Chvátal (1980), "Hard Knapsack Problems", *Operations Research* 28, 1402–1411.

[3] D. Fayard and G. Plateau (1994), "An exact algorithm for the 0-1 collapsing knapsack problem", *Discrete Applied Mathematics* 49, 175–187.

[4] S. Martello and P. Toth (1984), "A Mixture of Dynamic Programming and Branch-and-Bound for the Subset-Sum Problem", *Management Science* 30, 765–771.

[5] S. Martello and P. Toth (1988), "A New Algorithm for the 0-1 Knapsack Problem", *Management Science* 34, 633–644.

[6] S. Martello and P. Toth (1990), *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, Chichester.

[7] S. Martello and P. Toth (1997), "Upper Bounds and Algorithms for Hard 0-1 Knapsack Problems", *Operations Research* 45, 768–778.

[8] U. Pferschy, D. Pisinger, G.J. Woeginger (1997), "Simple but Efficient Approaches for the Collapsing Knapsack Problem", *Discrete Applied Mathematics* 77, 271–280.

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

[10] D. Pisinger (1996), "Strongly correlated knapsack problems are trivial to solve", *Proceedings CO96*, London, 27–29 March, 1996. Submitted for publication.

[11] D. Pisinger (1997), "A minimal algorithm for the 0-1 knapsack problem", *Operations Research* 45, 758–767.

[12] D. Pisinger, P. Toth (1998), "Knapsack Problems", to appear in: D.-Z. Du, P. Pardalos (eds.), *Handbook of Combinatorial Optimization*, Kluwer Academic Publishers.

[13] D. Pisinger (1998), "Core Problems in Knapsack Algorithms", *DIKU, University of Copenhagen,* Report 94/26. To appear in *Operations Research.*

[14] G. Plateau, M. Elkihel (1985), "A Hybrid Method for the 0-1 Knapsack Problem", *Methods of Operations Research* 49, 277–293.

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

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | 1 |
| 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 | — | — | 0.07 | 0.04 | 0 |

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

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | 1 |
| 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 | 0.09 | 1.19 | 0.00 | 0.01 | 0 |

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

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | 1 |
| 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 | 4.13 | — | 0.02 | 0.08 | 0 |

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

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | 1 |
| 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 | 0.01 | 0.01 | 0.00 | 0.01 | 0 |

Table 5: Max number of states (in thousands)

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc.sim |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0 | 0 | 0 | 0 | 4 | 27 | 3 | 38 | 3 | 6 | 6 | 39 | 2 | 25 | 2 | 22 | 0 |

Table 6: Average core size

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc.sim |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 11 | 10 | 17 | 18 | 26 | 23 | 26 | 23 | 28 | 25 | 12 | 15 | 18 | 16 | 26 | 24 | 23 |

Table 7: Number of relaxed problems solved

| | uncorr | | weak.corr | | str.corr | | inv.str.corr | | al.str.corr | | subset-sum | | ev.odd s.s. | | ev.odd kp | | uc.sim |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n \backslash R$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0 | 0 | 0 | 0 | 3 | 23 | 2 | 18 | 2 | 4 | 11 | 74 | 2 | 56 | 1 | 19 | 0 |

Table 8: Average solution times

| $n \backslash R$ | Avis sub.sum | Avis knapsack | collaps.kp | bou.str.corr | no small w |
|---|---|---|---|---|---|
| 50 | 0.26 | 0.00 | 0.02 | 0.00 | 0.00 |

* Could not be generated due to limited integer size