Available online at www.sciencedirect.com
ScienceDirect
European Journal of Operational Research 186 (2008) 63–76
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Discrete Optimization

A dynamic programming based reduction procedure for the multidimensional 0–1 knapsack problem

Stefan Balev [a,*], Nicola Yanev [b], Arnaud Fréville [c], Rumen Andonov [d]

[a] LITIS, Le Havre University, 25 rue Ph. Lebon BP 540, 76058 Le Havre Cedex, France
[b] Faculty of Mathematics and Computer Science, Sofia University, 5, J. Bouchier str., 1126 Sofia, Bulgaria
[c] LAMIH/ROI, University of Valenciennes, Le Mont Houy, 59313 Valenciennes Cedex 9, France
[d] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Received 22 January 2003; accepted 16 February 2006
Available online 13 February 2007

Abstract

This paper presents a preprocessing procedure for the 0–1 multidimensional knapsack problem. First, a non-increasing sequence of upper bounds is generated by solving LP-relaxations. Then, a non-decreasing sequence of lower bounds is built using dynamic programming. The comparison of the two sequences allows either to prove that the best feasible solution obtained is optimal, or to fix a subset of variables to their optimal values. In addition, a heuristic solution is obtained. Computational experiments with a set of large-scale instances show the efficiency of our reduction scheme. Particularly, it is shown that our approach allows to reduce the CPU time of a leading commercial software.

Keywords: Dynamic programming; Integer programming; Multidimensional knapsack problem; Variable reduction; Heuristics

1. Introduction

In this paper, we present a preprocesssing scheme for the 0–1 Multidimensional Knapsack Problem (MKP), which can be formulated as

$$\begin{aligned} \max \quad & \sum_{j\in N} c_j x_j \\ \text{s.t.} \quad & \sum_{j\in N} a_{ij} x_j \leqslant b_i, \quad i \in M, \\ & x_j \in \{0,1\}, \quad j \in N, \end{aligned}$$

where $N = \{1, 2, \ldots, n\}$ is the set of items, $M = \{1, 2, \ldots, m\}$ is the set of knapsack constraints with capacities b_i, associated weights a_{ij} and profits c_j. The objective is to find a subset of items that yields a maximum profit.

* Corresponding author. Tel.: +33 232 744 321; fax: +33 232 744 314.
E-mail addresses: stefan.balev@univ-lehavre.fr (S. Balev), choby@math.bas.bg (N. Yanev), arnaud.freville@univ-valenciennes.fr (A. Fréville), randonov@irisa.fr (R. Andonov).

0377-2217/$ - see front matter
doi:10.1016/j.ejor.2006.02.058

We assume that all the data a_{ij}, b_i, c_j are non-negative integers and, without loss of generality, that $c_j > 0$, $b_i > 0$, $a_{ij} \leqslant b_i$ for all $j \in N$ and all $i \in M$ and $\sum_{j \in N} a_{ij} > b_i$ for all $i \in M$.

The MKP is typically encountered in the areas of capital budgeting and resource allocation. The paper by Manne and Markowitz [27] is probably one of the earliest references to this problem. Other applications include project selection, cutting stock, loading problems, determining the optimal investment policy for the tourism sector of a developing country, and, more recently, delivery of groceries in vehicles with multiple compartments, approval voting, multi-project scheduling, satellite communications. It also appears in a collapsing problem and as a subproblem in large models for allocating processors and data bases in a distributed computer system. Finally, the MKP model is more and more frequently used as a benchmark to compare general purpose methods as metaheuristics.

In this paper we will often use shortcut notations for the problem like

$$\max\{cx : a_i x \leqslant b_i,\ i \in M,\ x \in B^n\},$$

or

$$\max\left\{\sum_{j \in N} c_j x_j : \sum_{j \in N} A_j x_j \leqslant b,\ x_j \in B,\ j \in N\right\},$$

or simply

$$\max\{cx : Ax \leqslant b,\ x \in B^n\}.$$

2. Related work

The multidimensional knapsack problem generalizes the well-known Knapsack Problem (KP) which deals with only one constraint. As the single constraint case, the MKP is $\mathcal{NP}$-hard but not strongly $\mathcal{NP}$-hard. Polynomial approximation schemes exist for the single knapsack problem and some of them are generalized for the MKP [6,13]. But while there are fully polynomial approximation schemes for $m = 1$, finding fully polynomial approximations for $m > 1$ is $\mathcal{NP}$-hard [15,26].

Most of the research on knapsack problems deals with the much simpler single constraint case ($m = 1$). This problem is very well studied and efficient exact and approximate algorithms have been developed for obtaining optimal and near-optimal solutions. An extensive overview of exact and heuristic algorithms is given by Martello and Toth [29]. Randomly generated instances up to 250 000 variables may be solved to optimality. Important recent advances can be found in [30,34].

2.1. Exact methods

In contrast, the MKP is significantly harder to solve in practice than the KP. As soon as the number of knapsack constraints increases, exact algorithms usually fail to provide an optimal solution of moderate size instances in a reasonable amount of time. For example, one of the recent versions of CPLEX (6.5.2) is not able to solve difficult problems with 100 variables and 5 constraints to optimality, because of the memory requirements of the search tree [33].

All general 0–1 integer programming techniques may be applied to the MKP [8,31,32]. Only the non-negativity of the coefficients distinguishes this problem from the general 0–1 integer programming problem. The dense constraint matrix and the absence of special constraints, such as generalized upper bounds, special-ordered sets, etc., complicate the development of efficient algorithms for the MKP. That is why relatively few special-purpose algorithms address the MKP.

The development of exact algorithms for 0–1 integer programming began several decades ago [2,3,16,18]. Typically, these approaches start with preprocessing phase, finding lower and upper bounds of the objective value and trying to reduce the problem size by fixation of variables and elimination of constraints. The second phase is an implicit enumeration, which uses the preprocessing information.

The first special-purpose branch and bound algorithm for the MKP is published by Shih [38]. An upper bound is obtained using the minimum of the LP-relaxation values associated with each of the m single

constraint knapsack problems. Gavish and Pirkul [14] develop another branch-and-bound algorithm using the surrogate relaxation of the problem. Their algorithm was proved to be faster than the Shih's one for randomly generated instances containing up to 200 variables and 5 constraints. Other interesting results are given by Fréville and Plateau [12] for the case of two constraints ($m = 2$). They solve instances up to 2000 variables by using an efficient preprocessing phase based on an exact solving of the surrogate dual.

Other kinds of special-purpose exact methods, with only limited success being reported, include the dynamic programming based methods [17,43], an enumerative algorithm based on the Fourier–Motzkin elimination [5] and an iterative scheme using linked LP-relaxations, disjunctive cuts and implicit enumeration [39].

2.2. Heuristic methods

A lot of heuristic methods were developed during the last three decades, the best of them being able to provide near-optimal solutions for instances with sizes up to $n = 500$ and $m = 30$, in a reasonable amount of CPU time. The main ideas developed for the MKP can be roughly classified into four categories. The greedy algorithms construct a feasible solution step by step by fixing one (or more) variables at each iteration. The first algorithm using the greedy principle was published by Senju and Toyoda [37]. Relaxation-based methods combine local search and information provided by the LP-relaxation, the Lagrangean relaxation or the surrogate relaxation, to generate feasible solutions (a comprehensive review of these methods is given by Fréville and Plateau [10]). The last two groups concern metaheuristics, which are very popular methods since the last decade. Simulated annealing, threshold method, tabu search and GRASP are used to enhance the basic local search by overcoming bad local optima. Particularly, good results have been obtained with tabu search embedded into a strategic oscillation scheme [19,23]. Finally, promising approaches were recently developed concerning genetic algorithms for constrained optimization [7].

2.3. Preprocessing techniques

Preprocessing techniques play a fundamental role in the development of efficient integer programming methods. The basic techniques try, among other things, to fix variables, to identify infeasibility and constraint redundancy, to tighten the LP-relaxation by modifying coefficients and by generating strong valid inequalities. Most of the research deals with branch-and-cut algorithms which were successful for solving large scale 0–1 linear programming problems. Seminal ideas to generate strong valid inequalities are given by Crowder et al. [8]. Savelsbergh [36] presents a framework of basic techniques to improve the representation of a mixed integer programming problem. Constraint pairing [22], probing techniques and logical reduction methods [21], are also very useful techniques to improve enumeration procedures and branch-and-cut methods. More recently, Glover et al. [20] generate cuts from surrogate constraint analysis. Constraint pairing and surrogate analysis are used by Osorio et al. [33] to generate logic cuts. Their computational experiments show that the preprocessing approach improves the performance of CPLEX.

The reduction technique presented in our paper tries to fix some of the variables to their optimal values. The common variable fixation techniques are based on a good lower bound $l = c\underline{x}$ associated with a feasible solution $\underline{x}$. They use the following property:

For any $j \in N$ and for any $\varepsilon \in \{0, 1\}$, if

$$z_j = \max\{cx : Ax \leqslant b,\ x \in B^n,\ x_j = \varepsilon\} \leqslant l,$$

then either $x_j = 1 - \varepsilon$ in any optimal solution of the MKP, or $\underline{x}$ is optimal.

Of course, the above MKP is as hard to solve as the original problem, but it should be clear that any upper bound on z_j suffices. Fayard and Plateau [9] use reduced costs associated with Lagrangean relaxation to derive such upper bounds. Fréville and Plateau [11] propose more refined upper bounds induced by additivity of the reduced costs and separation on the optimal basic fractional variable of surrogate relaxations.

In this paper we propose an alternative approach based on dynamic programming and LP upper bounds. The idea of this approach is contained in [42] for the single constraint case. We extend it for the MKP and introduce a number of improvements and new ideas.

3. Dynamic programming approaches

Dynamic programming (DP) was introduced by Bellman [4]. Toth presents the early DP-based approaches for the KP in [40] and reports numerical experiments with limited success. More recently, Pisinger proposes a DP algorithm for KP which constructs a core problem of minimal size, thus minimizing the sorting and reduction efforts. Hybrid methods, combining dynamic programming and implicit enumeration, were developed for the KP. The first approach was published by Plateau and Elkihel [35]. A recent approach, the so-called combo algorithm, is able to solve very large instances up to 10 000 variables within less than one second, with basically no difference in the solution times of "easy" and "hard" instances [30].

Marsten and Morin [28] proposed the first hybrid method for the MKP, which combines heuristic algorithms, dynamic programming and branch-and-bound approaches. More sophisticated methods, such as a successive sublimation procedure can be found in [25].

3.1. General method

Consider the function

$$f(k,g) = \max\left\{\sum_{j=1}^{k} c_j x_j : \sum_{j=1}^{k} A_j x_j \leqslant g,\ x_j \in \{0,1\},\ j = 1,\dots,k\right\}. \tag{1}$$

Obviously, $f(n,b)$ is the optimal value of the MKP. This value can be found using the recursion

$$f(k,g) = \max\{f(k-1,g), c_k + f(k-1,g-A_k)\}$$

for $k = 1,\dots,n$ and $A_k \leqslant g \leqslant b$ with boundary conditions

$$f(0,g) = 0, \quad 0 \leqslant g \leqslant b,$$
$$f(k,g) = f(k-1,g), \quad k = 1,\dots,n,\ g \geqslant 0,\ g \not\geqslant A_k.$$

From a practical point of view, to solve the MKP problem we have to fill a table of size $n \times b_1 \times \cdots \times b_m$ which can be very memory- and time-consuming even for medium-size instances of the single knapsack problem ($m = 1$). The first step to overcome the problem is to use a sparse representation of the table as described in the next section.

3.2. List representation

This kind of representation is used in [1,24,40] for single knapsack problems. It is based on the step-wise growing nature of the knapsack function and uses a list containing only the points where the value of the knapsack function changes. In this section we present a natural generalization of this idea for the MKP.

During the solution process we keep a list of $(m+1)$-tuples. Each tuple corresponds to a feasible solution. It represents the objective value and the residual capacities of this solution. We start with the list $\{(0,b)\}$ corresponding to the solution $x = \mathbf{0}$. At kth step, $k = 1,\dots,n$ we construct new solutions with $x_k = 1$ using the ones constructed at the $(k-1)$th step (if possible) and add them to the list. The process is formally described in Algorithm 1.

Algorithm 1. (*DP using list representation*)

```
1  L = (0, b)
2  for k = 1, ..., n
3     L1 = ∅
4     for (f, g) ∈ L
5        if g ⩾ A_k then L1 = L1 ∪ {(f + c_k, g − A_k)}
6     end for
7     L = L ∪ L1
8  end for
```

To restore the optimal solution, it suffices to keep in each element of the list a pointer to the element from which it was obtained or simply the step on which it was obtained.

Even for small problem instances the computational effort of the DP algorithm is considerable. This complexity is in the nature of the method because in fact it generates *all* the feasible solutions of the problem. Hence it is not much better than an explicit enumeration of all 0–1 vectors.

There are different strategies to recognize the solutions which are not perspective and to remove them from the list at earlier stages. The most straightforward technique is to evaluate the new solutions using bounds and then the dynamic programming becomes nothing but a kind of breadth-first branch-and-bound algorithm. We will show a different way to use bounds in Section 4.

Another way to reduce the size of the list is to use a dominance. This method uses the step-wise growing behavior of the knapsack function (1). It is especially suitable for single knapsack problems. Several variants of dominance for single 0–1 knapsack can be found in [1,24,40]. The main idea is very simple. Consider two elements of the list $e_1 = (f_1, g_1)$ and $e_2 = (f_2, g_2)$. Suppose that $f_1 \geqslant f_2$ and $g_1 \geqslant g_2$, that is the solution corresponding to e_1 has a better objective function and more residual capacity than the one corresponding to e_2. In this case it is clear that any continuation of the solution corresponding to e_1 will be no worse than any continuation of e_2. That is why we can remove e_2 from the list without missing the optimal solution. The dominance is very efficient for single knapsack problems ($m = 1$). Unfortunately this is not the case for $m > 1$, where detecting the dominance is very time-consuming and moreover, the dominance occurs very rarely in the solution process. The reason is that for each two numbers a and b, either $a \leqslant b$ or $b \leqslant a$, while for vectors we have not such a total order. Our preliminary computational experiments showed that it is not worthwhile to use dominance for the MKP, because the high price paid to detect it is not compensated by considerable elimination of elements of the list.

The analysis of the DP method shows that it is not directly applicable in the context of the MKP. In the next section we show a combination of this method with a bounding technique which allows to fix some of the variables at their optimal values and in many cases even to solve the entire problem.

4. A new procedure to fix variables

We propose a new method which combines LP-relaxation, upper bounds, and dynamic programming in order to obtain efficient reduction and heuristic procedures for the MKP.

As we have seen in the previous sections, the DP method is not directly applicable. Nevertheless the first several steps of the algorithm are very fast since there are no expensive computations. In this section we will show how to use this advantage.

4.1. Main results

Let $\underline{x}$ be a *feasible* solution of our problem

$$z = \max\{cx : Ax \leqslant b,\ x \in B^n\}, \tag{2}$$

and let u_j, $j = 1, \ldots, n$ be upper bounds of the problems where x_j is fixed at the opposite value of $\underline{x}_j$:

$$u_j \geqslant z_j = \max\{cx : Ax \leqslant b,\ x_j = 1 - \underline{x}_j,\ x \in B^n\}. \tag{3}$$

Let us reorder the variables so that

$$u_1 \geqslant u_2 \geqslant \cdots \geqslant u_n. \tag{4}$$

We introduce the variables in the DP algorithm using this order. At the end of each step of the algorithm we calculate lower bounds l_k by completing with $\underline{x}_{k+1}, \ldots, \underline{x}_n$ the best possible entry of the list L so that the obtained solution is feasible. To be more precise

$$l_k = \max_{(f,g)\in L} \left\{ f : g \geqslant \sum_{j=k+1}^{n} A_j \underline{x}_j \right\} + \sum_{j=k+1}^{n} c_j \underline{x}_j. \tag{5}$$

The meaning of the bounds l_k is explained by the next proposition (see [30] for proof in the case $m = 1$).

Proposition 1. *For each* $k = 1, \ldots, n$

$$l_k = \max\left\{cx : Ax \leqslant b,\ x \in B^n,\ x_j = \underline{x}_j,\ j = k+1, \ldots, n\right\}. \tag{6}$$

From this proposition it follows that

$$l_1 \leqslant l_2 \leqslant \cdots \leqslant l_n. \tag{7}$$

In this way we obtain a non-increasing sequence of upper bounds (4) and a non-decreasing sequence of lower bounds (7). At the moment when the both sequences "meet" each other we are done and the optimal solution is found as shows the next proposition.

Proposition 2. *If* $l_k \geqslant u_{k+1}$ *for some* k *then* $z = l_k$.

Proof. Let x^* be an optimal solution of (2).

1. Let $x^*_j = \underline{x}_j$ for all $j = k+1, \ldots, n$. Then from (2) and (6) the proposition is obvious.
2. Let $x^*_j = 1 - \underline{x}_j$ for some $j = k+1, \ldots, n$. Then

$$z = z_j \leqslant u_j \leqslant u_{k+1} \leqslant l_k.$$

On the other hand we have $z \geqslant l_k$, hence $z = l_k$. □

Algorithm 2 gives a formal description of the method. Note that if the list L is sorted by f then l (line 12) can be computed easier. The only essential complication comparing to Algorithm 1 comes from the lines 1–3.

Algorithm 2. (*DP with bounds*)

```
1   Find a feasible solution x
2   Compute the bounds u_1, ..., u_n
3   Reorder the variables to satisfy (4)
4   L = (0, b), p = cx, q = Ax
5   for k = 1, ..., n
6     L_1 = ∅
7     for (f, g) ∈ L
8       if g ⩾ A_k then L_1 = L_1 ∪ {(f + c_k, g − A_k)}
9     end for
10    L = L ∪ L_1
11    if x_k = 1 then p = p − c_k, q = q − A_k
12    l = p + max_{(f,g)∈L}{f : g ⩾ q}
13    if l ⩾ u_{k+1} then break
14  end for
```

Even in the version with bounds the list may become too long before we fix all the variables (this is in fact the common case). But the first several steps are very fast and they can help to fix many variables. In this way we can use the proposed method as an efficient preprocessing procedure which reduces the size of the problem instead of an exact algorithm. The next proposition explains the mechanism of fixing the variables.

Proposition 3. *If* $l_s \geqslant u_k$ *for some* $s < k$ *then there exists an optimal solution* x^* *of the MKP such that* $x^*_j = \underline{x}_j$, $j = k, \ldots, n$.

Proof. Consider l_{k-1} and the corresponding solution. Use $l_{k-1} \geqslant l_s \geqslant u_k$ and apply Proposition 2. □

The first question is how to choose the moment to stop, in other words, the number of the DP steps to perform. This number of steps, say s, is an algorithm parameter that can be used to tune the code according to the available memory size and the size of the problem. For our computational experiments we used $s = 18 - \lfloor \log_2(m+2) \rfloor$ and the time to perform the first s DP steps was practically 0. In general, when we choose the number of steps we have to consider the tradeoff between the speed of the algorithm and its quality. In the worst case each step is twice slower than the previous one. On the other hand, the bounds l_k become better at each step and we have the chance to fix more variables.

Another observation that allows to improve the code is that there is no need to compute the bounds l_k at each step. For large-size problems the chances to fix all the variables in several steps are small and that is why we prefer to perform the first s steps without computing the lower bounds (in this case the steps become faster) and to compute only the bound l_s after the sth step.

The last observation is that once we finish and fix some of the variables we can apply the same method for the reduced problem. Since part of the variables are fixed, the upper bounds for the reduced problem will be tighter and it is possible to fix some other portion of variables. We can continue in this way until we fix all the variables or no variable can be fixed.

The final version of our method is outlined in Algorithm 3.

Algorithm 3. (*DP with bounds* (*revised*))

Find a feasible solution $\underline{x}$
Compute the bounds $u_1, \ldots, u_n$
Reorder the variables to satisfy (4)
Compute the number of DP steps s
$L = (0, b)$
`for` $k = 1, \ldots, s$
  $L_1 = \emptyset$
  `for` $(f, g) \in L$
    `if` $g \geqslant A_k$ `then` $L_1 = L_1 \cup \{(f + c_k, g - A_k)\}$
  `end for`
  $L = L \cup L_1$
`end for`
$l = \sum_{j=s+1}^{n} c_j \underline{x}_j + \max_{(f,g)\in L}\{f : g \geqslant \sum_{j=s+1}^{n} A_j \underline{x}_j\}$
`if` $l < u_n$ `then stop` (no fixation is possible)
`else if` $l \geqslant u_{s+1}$ `then stop` (all the variables are fixed)
`else`
  let k be such that $u_{k-1} > l \geqslant u_k$
  fix $x_j = \underline{x}_j$, $j = k, \ldots, n$
  apply the same algorithm for the reduced problem
`end if`

4.2. Computing the bounds

In Section 4.1 we give a general framework of the method without considering how to generate the feasible solution $\underline{x}$ or the upper bounds u_j. The choice of the initial feasible solution and the method of computing the upper bounds are crucial for the performance of our algorithm. A better initial solution and tighter bounds give the possibility to fix more variables, but on the other hand they are more computationally expensive.

The most commonly upper bounds used in integer programming are the ones produced by linear programming (LP). It seems that they have the best quality/complexity tradeoff. That is why we use this kind of bounds. More precisely

$$u_j = \lfloor \max\{cx : Ax \leqslant b,\ \mathbf{0} \leqslant x \leqslant \mathbf{1},\ x_j = 1 - \underline{x}_j\} \rfloor. \tag{8}$$

The next question is how to choose the feasible solution $\underline{x}$. The choice of LP bounds forces the use of the LP solution. Let $\bar{x}$ be the optimal solution of the LP-relaxation of the MKP

$$\max\{cx : Ax \leqslant b,\ \mathbf{0} \leqslant x \leqslant \mathbf{1}\}, \tag{9}$$

and let $u = \lfloor c\bar{x} \rfloor$. This solution contains m basic variables which are typically fractional and the rest $n - m$ variables are 0–1. Now suppose that for some j with integer $\bar{x}_j$ we choose $\underline{x}_j = 1 - \bar{x}_j$. Then the bound u_j will be equal to u and the fixation of the variable x_j will never work. That is why the only reasonable choice is $\underline{x}_j = \bar{x}_j$ whenever $\bar{x}_j$ is integer. For the fractional variables the only restriction is that $\underline{x}$ must be feasible. In the other hand, we are interested to have a good feasible solution. Let $\mathscr{B}$ be the set of the basic (fractional) variables. We can consider the restricted problem

$$\max\{cx : Ax \leqslant b,\ x_j = \bar{x}_j,\ j \notin \mathscr{B},\ x_j \in \{0,1\},\ j \in \mathscr{B}\},$$

with only m variables and less right-hand sides. Since m is assumed to be small we can solve it exactly (using for example the simplest variant of DP) and take the optimal solution as values for $\underline{x}_j$, $j \in \mathscr{B}$.

Finding the bounds u_j, $j = 1, \ldots, n$ involves solving n linear programs which can be time-consuming process. But we can use the solution of (9) as a starting point for the optimization of (8). In this case only a small number of simplex iterations is needed and finding the bounds is very fast.

Example 1. Consider the following instance of the MKP with 10 variables and 2 constraints

$$\left(\begin{array}{c|c} c & \\ \hline A & b \end{array}\right) = \left(\begin{array}{cccccccccc|c} 31 & 92 & 53 & 36 & 44 & 43 & 54 & 44 & 42 & 46 & \\ \hline 19 & 83 & 99 & 56 & 76 & 91 & 62 & 89 & 95 & 16 & 290 \\ 42 & 93 & 49 & 60 & 2 & 8 & 38 & 3 & 24 & 58 & 200 \end{array}\right).$$

The solution of its LP-relaxation is

$$\bar{x} = (0.1744, 1, 0, 0, 1, 0, 1, 0.5582, 0, 1).$$

The only solution of the restricted problem

$$\max\{31x_1 + 44x_8 : 19x_1 + 89x_8 \leqslant 53, 42x_1 + 3x_8 \leqslant 9,\ x_1, x_8 \in \{0,1\}\},$$

is $(0, 0)$ and hence the initial feasible solution is

$$\underline{x} = (0, 1, 0, 0, 1, 0, 1, 0, 0, 1).$$

The upper bounds are

$$(u_1, \ldots, u_{10}) = (264, 240, 243, 241, 257, 257, 260, 262, 246, 249),$$

which imposes the following order of the variables

$$x_1, x_8, x_7, x_5, x_6, x_{10}, x_9, x_3, x_4, x_2.$$

Table 1 gives the list L, p, q and the lower bound l at the end of each phase of the algorithm. The tuple $(p; q)$ is placed next to the list element with which it is combined to obtain the lower bound. After the third iteration of the loop in lines 6–12 of Algorithm 3 we stop because the rest of the upper bounds are less or equal to the current lower bound. The optimal solution is already known. It is

$$x^* = (1, 1, 0, 0, 1, 0, 0, 1, 0, 1)$$

with objective value 257.

It is easy to show that our reduction scheme is stronger than previous methods based on reduced costs only [11,23]. Let $\overline{v}$ be the optimal values of the dual variables associated to the knapsack constraints. Let $r_0 = \sum_{i \in M} \overline{v}_i b_i$, $r_j = \sum_{i \in M} \overline{v}_i a_{ij}$ for $j \in N$,

Table 1
An example of solving the MKP by Algorithm 3

Var	L	$(p;q)$	l
x_1	$(0;290,200)$ $(31;271,158)$	$(236;237,191)$	236
x_8	$(0;290,200)$ $(31;271,158)$ $(44;201,197)$ $(75;182,155)$	$(236;237,191)$	236
x_7	$(0;290,200)$ $(31;271,158)$ $(44;201,197)$ $(54;228,162)$ $(75;182,155)$ $(85;209,120)$ $(98;139,159)$ $(129;140,117)$	$(182;175,153)$	257

$$S(\overline{v}) : \max\left\{\sum_{j\in N} c_j x_j : \sum_{j\in N} r_j x_j \leqslant r_0,\ \mathbf{0} \leqslant x \leqslant \mathbf{1}\right\},$$

be the surrogate relaxation with relaxed integrality constraints and let

$$LR(\lambda) : \lambda r_0 + \max\left\{\sum_{j\in N}(c_j - \lambda r_j)x_j : x \in B^n\right\},$$

be the Lagrangean relaxation of $S(\overline{v})$. It is well known that, first $z(LR(1)) = z(S(\overline{v})) = c\overline{x}$, and second, that if $\lfloor z(LR(1) : x_j = 1 - \underline{x}_j)\rfloor = \lfloor z(LR(1))\rfloor + (c_j - r_j) \leqslant c\underline{x}$ for any j, such that $c_j - r_j < 0$, then the variable x_j can be fixed to $\underline{x}_j$ (there is a symmetric result for $c_j - r_j > 0$). As $u_j \leqslant \lfloor z(LR(1) : x_j = 1 - \underline{x}_j)\rfloor$ (since $\lambda = 1$ is not necessarily the optimal multiplier for the reduced problem), our reduction test $u_j \leqslant c\underline{x}$ is stronger than the ones using the reduced costs only.

Example 2. Consider the following example ($n = 15$, $m = 4$) from [33]

$$\left(\begin{array}{rrrrrrrrrrrrrrr|r}
36 & 83 & 59 & 71 & 43 & 67 & 23 & 52 & 93 & 25 & 67 & 89 & 60 & 47 & 64 & \\
\hline
7 & 19 & 30 & 22 & 30 & 44 & 11 & 21 & 35 & 14 & 29 & 18 & 3 & 36 & 42 & 87 \\
3 & 5 & 7 & 35 & 24 & 31 & 25 & 37 & 35 & 25 & 40 & 21 & 7 & 17 & 22 & 75 \\
20 & 33 & 17 & 45 & 12 & 21 & 20 & 2 & 7 & 17 & 21 & 11 & 11 & 9 & 21 & 65 \\
15 & 17 & 9 & 11 & 5 & 5 & 12 & 21 & 17 & 10 & 5 & 13 & 9 & 7 & 13 & 55
\end{array}\right).$$

Our method provides the initial solution $\underline{x}$ with objective value $c\underline{x} = 301$ as shown in second row of Table 2 (the same initial solution is used in [33]). The reduced costs of the LP-relaxation allow to fix only the variables x_7, x_8 and x_{10}. Our upper bounds u_j allow to fix 8 variables at their optimal values (in italic on the third row of Table 2). If we fix these variables and recompute the upper bounds, three more variables may be fixed. After one more iteration only x_3 is not fixed, but one can easily see that its optimal value is 1. In this way $\underline{x}$ is proved to be an optimal solution.

5. Experimental results

The implementation of Algorithm 3 is written in C. To obtain a faster code and to compare with the last achievements in 0–1 programming we use the commercial product CPLEX of ILOG. The code is compiled

Table 2
Upper bounds for Example 2

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\underline{x}_j$	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0
Red. costs	311	335	335	315	324	330	*292*	*294*	335	*299*	335	312	313	325	311
u_j^1	308	327	332	*232*	311	*294*	*277*	*277*	309	*285*	*285*	*299*	304	306	*295*
u_j^2	*301*	325	328		311				*290*				*300*	306	
u_j^3		$-\infty$	326		$-\infty$									$-\infty$	

using `gcc` v. 2.96 with option `-O3` and linked to CPLEX callable library. The computational experiments are performed on Compaq AlphaServer DS20 with EV6/500 MHz processor.

We tested our approach on several data sets. For the first experiment we generated random instances, following the procedure described in [11]. To generate the coefficients of the matrix A we use two types of probability distributions. The first one is the uniform $U(0,M)$ distribution. The second one, which we call $D(\alpha,p,M)$, has a density function

$$f(x)=\begin{cases}\lambda-\frac{\lambda-\theta}{\alpha}x & \text{if } 0\leqslant x\leqslant\alpha,\\ \frac{\theta}{M-\alpha}(M-x) & \text{if } \alpha\leqslant x\leqslant M,\\ 0 & \text{otherwise.}\end{cases}$$

The parameters θ and λ are chosen so that $F(\alpha)=p$ and $F(M)=1$ where F is the distribution function.

In both cases the objective function c and the right-hand side b are generated in the following way:

$$c_j=\frac{\sum_{i=1}^{m}A_{ij}}{m}+500r_j,\quad r_j\in U(0,1),\ \ j=1,\ldots,n, \tag{10}$$

$$b_i=r_i\sum_{j=1}^{n}A_{ij},\quad r_i\in U(0,1),\ \ i=1,\ldots,m. \tag{11}$$

Table 3
Reduction results

Distribution	m	n	Reduced n (DP algorithm)				F and P
			Min	Max	Avg	Avg %	Avg %
$U(0,1000)$	5	100	0	49	15.1	84.9	63.6
		250	20	100	37.2	85.1	62.9
		500	0	51	21.4	95.7	68.1
	10	100	0	39	20.5	79.5	42.0
		250	0	68	17.2	93.1	49.4
		500	0	170	49.4	90.1	56.0
$D(\alpha,p,M)$	5	100	0	94	28.3	71.7	45.0
$\alpha=100$		250	16	197	57.8	76.9	61.7
$p=0.9$		500	0	403	111.8	77.6	60.9
$M=1000$	10	100	0	78	28.1	71.9	19.8
		250	0	230	73.7	70.5	27.9
		500	0	243	82.6	83.5	34.8
$D(\alpha,p,M)$	5	100	0	94	44.9	55.1	56.6
$\alpha=50$		250	45	234	138.9	44.4	44.2
$p=0.9$		500	0	492	144.4	71.1	46.8
$M=1000$	10	100	0	90	33.2	66.8	13.4
		250	0	217	116.5	53.4	12.0
		500	0	451	152.1	69.6	22.3

Table 4
Time and gap results

Distribution	m	n	Average time (milliseconds)				r	Gap %		
			t_1	t_2	t_3	t_4		Min	Max	Avg
$U(0,1000)$	5	100	78	55	133	153	13.0	0	0.10	0.02
		250	165	298	463	563	17.8	0	0.28	0.05
		500	333	420	753	2187	65.5	0	0.01	0.00
	10	100	92	87	178	208	14.4	0	0.06	0.01
		250	165	417	582	1287	54.8	0	0.08	0.01
		500	465	11,678	12,143	26,648	54.4	0	0.08	0.02
$D(\alpha,p,M)$	5	100	97	25	122	60	−102.8	0	0.84	0.10
$\alpha = 100$		250	177	170	347	353	1.9	0	0.29	0.05
$p = 0.9$		500	427	1527	1953	2427	19.5	0	0.18	0.03
$M = 1000$	10	100	137	42	178	118	−50.7	0	0.52	0.05
		250	273	243	517	607	14.8	0	0.42	0.09
		500	508	1547	2055	3720	44.8	0	0.42	0.07
$D(\alpha,p,M)$	5	100	103	42	145	63	−128.9	0	0.94	0.16
$\alpha = 50$		250	203	307	510	342	−49.3	0	0.75	0.27
$p = 0.9$		500	438	465	903	1048	13.8	0	0.38	0.07
$M = 1000$	10	100	130	93	223	173	−28.8	0	1.92	0.30
		250	257	413	670	817	18.0	0	0.55	0.18
		500	588	980	1568	1750	10.4	0	0.51	0.13

t_1 – Time of DP algorithm; t_2 – time to solve the reduced problem by CPLEX; t_3 – total time to solve the problem, $t_3 = t_1 + t_2$; t_4 – time to solve the initial problem by CPLEX; r – average reduction of the total solution time (in %), $r = \frac{t_4 - t_3}{t_4}$; gap – the gap between the solution given by DP and the optimal solution (in %), gap $= \frac{z - l_x}{z}$.

Table 3 gives the reduction results on this set of instances. Each row summarizes 10 instances. We give the minimum, maximum and average number of variables rest after the application of the DP algorithm. The average percentage of reduction is also given. For comparison we include the same percentage reported by Fréville and Plateau in [11] after applying their reduction procedure.

In 44 of the 180 instances the problem is completely solved by the DP algorithm. A considerable reduction of the problem size is also observed for the remaining cases.

Table 4 shows the efficiency of the algorithm in terms of CPU time and quality of the solution over the same set of instances. To estimate the efficiency, we apply the DP algorithm and then solve the reduced problem using CPLEX. We compare this total solution time with the time to solve the initial problem by CPLEX.

Table 4 shows that a negative reduction of CPU times occurs for some of the smaller instances (that is, solving the problem by DP is slower than solving it by general methods). However, when n increases, the effect

Table 5
Results for the data set of Chu and Beasley

m		5			10			30		
n	α	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
100	.25	20.2	9	0.18	2.1	1	0.60	0.0	–	0.61
100	.50	21.5	9	0.09	0.1	5	0.29	0.1	–	0.20
100	.75	30.7	13	0.07	10.6	9	0.09	0.4	−3	0.12
250	.25	22.0	–	0.23	0.8	–	0.35	0.0	–	0.23
250	.50	21.1	–	0.09	0.5	–	0.18	0.0	–	0.14
250	.75	42.0	17	0.04	10.6	–	0.08	0.0	–	0.09
500	.25	23.5	–	0.12	2.4	–	0.22	0.0	–	0.09
500	.50	22.3	–	0.06	1.4	–	0.09	0.0	–	0.05
500	.75	47.8	–	0.02	7.3	–	0.05	0.0	–	0.03

(1) – Reduction of the number of variables (in %), (2) – reduction of the total solution time (in %), (3) – gap between the best solution and the solution found by DP (in %).

of the DP algorithm becomes obvious. Looking at the reduction of the problem size, one can expect greater reduction of the solution time. But as it is well known, the reduced (also called core) problem inherits most of the "complexity" of the original problem.

Another measure of efficiency is the quality of the DP solution. Recall that when DP algorithm stops, we have a feasible solution (it's value being l_s) even if it is not able to fix all of the variables. Table 4 also gives the relative gap between this solution and the optimal solution of the problem. For more than the half (95) of the cases this gap is 0. For none of the cases the gap is more than 2%.

For the second experiment we use the data set of Chu and Beasley [7], available at http://msmga.ms.ic.ac.uk/. The matrix coefficients in this set are drawn from uniform $U(0, 1000)$ distribution. The objective coefficients are the same as in (11) and the right-hand side coefficients are the sum of the matrix coefficients in the corresponding row, multiplied by some constant α. For each $m = 5, 10, 30$, $n = 10, 250, 500$, and $\alpha = 0.25, 0.5, 0.75$ there are 10 instances, or a total of 270 instances. The summarized results for this data set are presented in Table 5.

The results for the last data set show that the performance of our method is particularly good for problem instances with a small number of constraints and a big number of variables. To confirm this observation we generated random instances with $m = 2$, n up to 6000, objective coefficients determined by (11) and right-hand sides $b_i = 0.25\sum_{j\in N} a_{ij}$. Table 6 shows that the reduction of the time grows with the size of the problem (for example the total solution time with reduction is about seven times less than the total solution time without reduction for $n = 4600$). For $n > 4600$ it is impossible to solve the problem without applying the reduction procedure (the search tree generated by CPLEX reaches the memory limit), while the total solution time when applying the reduction procedure is less than 1 minute. Observe also that the time t_3 (DP + CPLEX) has rel-

Table 6
Time results for $m = 2$, $\alpha = 0.25$

n	t_1	t_2	t_3	t_4	n	t_1	t_2	t_3	t_4
200	0.17	0.11	0.28	0.28	3200	8.47	4.70	13.17	69.97
400	0.30	0.28	0.57	1.55	3400	9.71	17.44	27.15	190.22
600	0.49	0.62	1.11	3.51	3600	11.02	10.36	21.38	122.90
800	0.76	0.92	1.68	8.35	3800	12.87	13.72	26.59	254.01
1000	1.11	1.57	2.68	14.84	4000	13.73	17.23	30.96	264.50
1200	1.45	1.54	2.99	13.29	4200	14.84	14.47	29.31	215.03
1400	1.88	3.13	5.01	26.96	4400	16.15	13.70	29.86	226.67
1600	2.49	2.14	4.63	25.67	4600	18.08	10.59	28.66	268.94
1800	3.32	3.96	7.28	52.81	4800	20.03	7.23	27.26	–
2000	3.60	3.91	7.51	44.18	5000	21.84	14.95	36.79	–
2200	4.16	2.44	6.60	34.14	5200	22.68	25.13	47.81	438.30
2400	5.12	4.00	9.11	61.70	5400	25.32	25.62	50.95	–
2600	6.03	5.48	11.50	74.64	5600	25.86	19.71	45.57	–
2800	6.51	8.04	14.56	90.09	5800	28.88	11.81	40.69	–
3000	7.72	10.46	18.18	133.40	6000	30.18	29.72	59.89	–

t_1 – Time of DP algorithm; t_2 – time to solve the reduced problem by CPLEX; t_3 – total time to solve the problem, $t_3 = t_1 + t_2$; t_4 – time to solve the initial problem by CPLEX. The times are in seconds.

Table 7
Results for the data set of Glover and Kochenberger

Problem	$n \times m$	Found by DP	Best known
GK018	100×25	4520	4528
GK019	100×25	3860	3869
GK020	100×25	5175	5180
GK021	100×25	3194	3200
GK022	100×25	2517	2523
GK023	200×15	9226	9235
GK024	500×25	9054	9070

atively robust behavior and grows slowly with n, while the time t_4 (CPLEX only) fluctuates and grows much faster.

Finally, we tested our approach on the data set of Glover and Kochenberger [19], a set of difficult instances, for which the optimal solutions are unknown. Our algorithm fails to reduce the size of these problems, but at least we quickly obtain good feasible solutions. Table 7 shows the objective values of the solutions we found and the best known solutions [41]. The time to obtain these solutions is less than 0.1 seconds for all the cases, while the times reported in [41] are considerable (about 300 seconds for 100 variables, 700 seconds for 250 variables, and 2000 seconds for 500 variables).

6. Conclusion

In this paper we presented a new dynamic programming based approach to the MKP. We introduce and use sparse data representation, which decreases memory and time requirements. We use the dynamic programming and the LP-relaxation information to derive lower and upper bounds allowing to find the optimal values of some or all the variables. The proposed algorithm is a fast and efficient preprocessing procedure allowing to reduce the problem size and in many cases even to find the optimal solution. Even if no variables are fixed, the procedure remains a robust and very fast heuristic method, providing a feasible solution of good quality by successive improvements of the rounded LP solution. We use the LP bounds but another bounding technique as Lagrangean or surrogate relaxation may be directly plugged in our algorithm.

The experimental results are promising and motivate future work on the improvement of the method. After the end of the DP phase, it is possible to make an attempt to reduce the gap between the lower and the upper bounds by tightening the upper bounds of the last variables using for example partial enumeration. Another reduction of the gap may come from improving the lower bound.

References

[1] J.H. Ahrens, G. Finke, Merging and sorting applied to 0–1 knapsack problem, Operations Research 23 (1975) 1099–1109.
[2] E. Balas, An additive algorithm for solving linear programs with zero–one variables, Operations Research 13 (1965) 517–546.
[3] E. Balas, Discrete programming by the filter method, Operations Research 19 (1967) 915–957.
[4] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, 1957.
[5] A.V. Cabot, An enumeration algorithm for knapsack problems, Operations Research 18 (1970) 306–311.
[6] A.K. Chandra, D.S. Hirshberg, C.K. Wong, Approximate algorithms for some generalized knapsack problems, Theoretical Computer Science 3 (1976) 293–304.
[7] P. Chu, J. Beasley, A genetic algorithm for the multidimensional knapsack problem, Journal of Heuristics 4 (1998) 63–86.
[8] H. Crowder, E.L. Johnson, H.W. Padberg, Solving large scale zero–one linear programming problems, Operations Research Society 31 (1983) 803–834.
[9] D. Fayard, G. Plateau, Reduction algorithm for single and multiple constraints 0–1 linear programming problems, in: Proceedings of Congress Methods of Mathematical Programming, Zakopane, 1977.
[10] A. Fréville, G. Plateau, Heuristics and reduction methods for multiple constraints 0–1 linear programming problems, European Journal of Operational Research 24 (1986) 206–215.
[11] A. Fréville, G. Plateau, An efficient preprocessing procedure for the multidimensional knapsack problem, Discrete Applied Mathematics 49 (1994) 189–212.
[12] A. Fréville, G. Plateau, The 0–1 bidimensional knapsack problem: Towards an efficient high-level primitive tool, Journal of Heuristics 2 (1996) 147–167.
[13] A.M. Frieze, M.R. Clarke, Approximation algorithms for the m-dimensional 0–1 knapsack problem: Worst-case and probablistic analyses, European Journal of Operational Research 15 (1984) 100–109.
[14] B. Gavish, H. Pirkul, Efficient algorithms for solving zero–one multidimensional knapsack problems to optimality, Mathematical Programming 31 (1985) 78–105.
[15] G.V. Gens, E.V. Levner, Computational complexity of approximation algorithms for combinatorial problems, in: Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. 74, 1979, pp. 292–300.
[16] A. Geoffrion, An improved implicit enumeration approach for integer programming, Operations Research 17 (1969) 437–454.
[17] P.C. Gilmore, R.E. Gomory, The theory and computation of knapsack functions, Operations Research 14 (1966) 1045–1074.
[18] F. Glover, A multiphase dual algorithm for the zero–one integer programming problem, Operations Research 13 (1965) 879–919.
[19] F. Glover, G. Kochenberger, Critical event tabu search for multidimensional knapsack problems, in: I. Osman, J. Kelly (Eds.), Metaheuristics: Theory and Applications, Kluwer Academic Publishers, 1996, pp. 407–427.
[20] F. Glover, H. Sherali, Y. Lee, Generating cuts from surrogate constraint analysis for zero–one and multiple choice programming, Computational Optimization and Applications 8 (1997) 151–184.

[21] M. Guignard, K. Spielberg, Logical reduction methods in zero–one programming minimal preferred variables, Operations Research 29 (1981) 49–74.
[22] P. Hammer, M. Padberg, U. Peled, Constraint pairing in integer programming, INFOR 13 (1975) 68–81.
[23] S. Hanafi, A. Fréville, An efficient tabu search approach for 0–1 multidimensional knapsack problem, European Journal of Operational Research 106 (1998) 659–675.
[24] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of ACM 21 (1974) 277–292.
[25] T. Ibaraki, Enumerative approaches to combinatorial optimization – Part II, Annals of Operations Research 11 (1987) 397–439.
[26] B. Korte, R. Schrader, On the existence of fast approximation schemes, in: O.L. Mangasarian, R.R. Meyer, S.M. Robinson (Eds.), Nonlinear Programming, 4, Academic Press, 1980, pp. 415–437.
[27] A.S. Manne, H.M. Markowitz, On the solution of discrete programming problems, Econometrica 25 (1957) 84–100.
[28] R.E. Marsten, T.L. Morin, A hybrid approach to discrete mathematical programming, Mathematical Programming 14 (1978) 21–40.
[29] S. Martello, P. Toth, Knapsack problems: Algorithms and computer implementationsSeries in Discrete Mathematics and Optimization, Wiley Interscience, New York, 1990.
[30] S. Martello, D. Pisinger, P. Toth, New trends in exact algorithms for the 0–1 knapsack problem, European Journal of Operational Research 123 (1999) 325–336.
[31] G.L. Nemhauser, M.W.P. Savelsbergh, G.C. Sigismondi, MINTO, a Mixed INTeger Optimizer, Operations Research Letters 15 (1994) 47–58.
[32] G.L. Nemhauser, L.A. Wolsey, Integer and combinatorial optimization. Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley, 1988.
[33] M.A. Osorio, F. Glover, P. Hammer, Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions, Research report, University of Puebla, Mexico, 2000, Presented at IFORS, Mexico, September 2000.
[34] D. Pisinger, A minimal algorithm for the 0–1 knapsack problem, Operations Research 45 (1997) 758–767.
[35] G. Plateau, M. Elkihel, A hybrid method for the 0–1 knapsack problem, Methods of Operations Research 49 (1985) 277–293.
[36] M.W.P. Savelsbergh, Preprocessing and probing techniques for mixed integer programming problems, ORSA Journal of Computing 6 (1994) 445–454.
[37] S. Senju, Y. Toyoda, An approach to linear programming with 0–1 variables, Management Science 15 (1968) 196–207.
[38] W. Shih, A branch and bound method for the multiconstraint knapsack problem, Journal of the Operational Research Society 30 (1979) 369–378.
[39] A.L. Soyster, B. Lev, W. Slivka, Zero–one programming with many variables and few constraints, European Journal of Operational Research 2 (1978) 195–201.
[40] P. Toth, Dynamic programming algorithms for the zero–one knapsack problem, Computing 25 (1980) 29–45.
[41] M.Vasquez, J.K. Hao, A hybrid approach for the 0–1 multidimensional knapsack problem, in: Proceedings of IJCAI-01, Seventeenth International Joint Conference on Artificial Intelligence, 2001, pp. 328–333.
[42] F.Viader, Méthodes de Programmation Dynamique et de Recherche Arborescente pour l'Optimisation Combinatoire: Utilisation Conjointe des deux approaches et Parallélisation d'Algorithmes, Ph.D. thesis, Univérsité Paul Sabatier, Toulouse, 1998.
[43] H.M. Weingartner, D.N. Ness, Method for the solution for the multidimensional 0–1 knapsack problem, Operations Research 15 (1967) 83–103.