Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Operations Research 32 (2005) 2271–2284
computers & operations research
www.elsevier.com/locate/dsw

# Where are the hard knapsack problems?

David Pisinger*

*Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark*

Available online 20 May 2004

**Abstract**

The knapsack problem is believed to be one of the "easier" $\mathcal{NP}$-hard problems. Not only can it be solved in pseudo-polynomial time, but also decades of algorithmic improvements have made it possible to solve nearly all standard instances from the literature. The purpose of this paper is to give an overview of all recent exact solution approaches, and to show that the knapsack problem still is hard to solve for these algorithms for a variety of new test problems. These problems are constructed either by using standard benchmark instances with larger coefficients, or by introducing new classes of instances for which most upper bounds perform badly. The first group of problems challenge the dynamic programming algorithms while the other group of problems are focused towards branch-and-bound algorithms. Numerous computational experiments with all recent state-of-the-art codes are used to show that (KP) is still difficult to solve for a wide number of problems. One could say that the previous benchmark tests were limited to a few highly structured instances, which do not show the full characteristics of knapsack problems.


*Keywords:* Knapsack problem; Dynamic programming; Branch-and-bound; Test instances

## 1. Introduction

The classical knapsack problem is defined as follows: We are given a set of $n$ items, each item $j$ having an integer profit $p_j$ and an integer weight $w_j$. The problem is to choose a subset of the items such that their overall profit is maximized, while the overall weight does not exceed a given capacity $c$. We may formulate the model as the following integer programming model:

$$\text{(KP)} \quad \text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1}$$

* Tel.: +45-35-32-1354; fax: +45-35-32-1401.
*E-mail address:* pisinger@diku.dk (D. Pisinger).

0305-0548/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2004.03.002

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leqslant c, \tag{2}$$

$$x_j \in \{0,1\}, \quad j = 1,\ldots,n, \tag{3}$$

where the binary decision variables $x_j$ are used to indicate whether item $j$ is included in the knapsack or not. Without loss of generality it may be assumed that all profits and weights are positive, that all weights are smaller than the capacity $c$, and that the overall weight of the items exceeds $c$.

From practical experience it is known that many (KP) instances of considerable size can be solved within reasonable time by exact solution methods. This fact is due to several algorithmic refinements which emerged during the last two decades. These include *advanced dynamic programming recursions*, the concept of solving a *core*, and the separation of *cover inequalities* to tighten the formulation. For a recent survey of the latest techniques see Martello et al. [1] or the monograph by Kellerer et al. [2].

The knapsack problem is $\mathcal{NP}$-hard in the weak sense, meaning that it can be solved in pseudo-polynomial time through dynamic programming. Meyer auf der Heide [3] showed that for the linear decision tree model (LDT) of computation no super-polynomial lower bound can exist. This negative result was extended by Fournier and Koiran [4] who showed that for even less powerful models of computation no super-polynomial lower bound is likely to exist.

The lack of theoretical upper and lower bounds on the computational complexity of (KP) leaves plenty of space for practical algorithmic development. Although none of these algorithms can ensure efficient solution times for all instances, progress is made on reaching acceptable running times for all "practically occurring" instances. In the following section, we will give a short overview of the latest exact algorithms for (KP), and state their worst-case complexity where possible. In Section 3, we experimentally measure the performance of these algorithms for a large variety of instance types, including two groups of new, difficult instances. The first group of difficult instances is based on large coefficients, while the second group contains six categories of structurally difficult instances with small coefficients. Although the classical problem instances are quite easy to solve for the most recent algorithms, it is interesting to see that the instances do not need to be changed much before the algorithms get a significantly different performance. In our search for algorithms, which are able to solve all "practically occurring" instances, it is important to be aware of these problems, and to extend our algorithms and in particular upper bounds to more robust variants. These thoughts are summarized in Section 4.

## 2. Exact algorithms for the knapsack problem

For the following discussion we need a few definitions. Assume that the items are sorted according to non-increasing efficiencies $p_j/w_j$, so that we have

$$\frac{p_1}{w_1} \geqslant \frac{p_2}{w_2} \geqslant \cdots \geqslant \frac{p_n}{w_n}. \tag{4}$$

The LP-relaxation of (KP) can be solved through the greedy algorithm by simply filling the knapsack until item $s = \min\{h\colon \sum_{j=1}^{h} w_j > c\}$, which is also known as the *split item*. The LP-bound is

then defined as

$$U_1 = \sum_{j=1}^{s-1} p_j + \left( c - \sum_{j=1}^{s-1} p_j \right) \frac{p_s}{w_s}.$$

The split item $s$ and hence also the LP-solution can be found in $O(n)$ time using a median-search algorithm presented by Balas and Zemel [5]. In the following, we will denote $z^*$ the integer optimal solution value and $x^*$ the corresponding solution vector. The *greedy solution*, choosing items $1,\ldots,s-1$, will be denoted $x'$.

Various branch-and-bound algorithms for (KP) have been presented. The more recent of these solve a *core* problem, i.e. a (KP) defined on a subset of the items where there is a large probability of finding an optimal solution. The MT2 algorithm [6] is the most advanced of these algorithms. It starts by solving the core problem obtaining a lower bound $z$ on the (KP) as well as an upper bound $U$. If $z = U$ it stops, otherwise it reduces the size of the instance by fixing variables at their optimal value. In the last phase, the reduced (KP) is solved to optimality.

Realizing that the core size is difficult to estimate in advance, Pisinger [7] proposed to use an expanding core algorithm, which simply starts with a core consisting of the split item only, and then adds more items to the core when needed. The proposed Expknap makes use of branch-and-bound where computationally cheap upper bounds from LP-relaxation are used.

Martello and Toth [8] proposed a special variant of the MT2 algorithm which was developed to deal with hard knapsack problems. The resulting algorithm MThard makes use of a new family of upper bounds based on the generation of additional *cardinality constraints* which again are Lagrangian relaxed to reach an ordinary (KP).

Straightforward use of dynamic programming leads to the well-known *Bellman recursion* [9] which solves the (KP) in pseudo-polynomial time $O(nc)$. Reversing the roles of profits and weights in the dynamic programming recursion leads to an algorithm with running time $O(nz^*)$.

Pisinger [10] presented an improved variant of the Bellman recursion which runs in time $O(nm/\log m)$ on a word RAM, where $m = \max\{c, z^*\}$. For the most difficult instances where $c$ and $z^*$ are of the same magnitude, this leads to a logarithmic improvement over the Bellman recursion.

A balanced solution is a solution which can be reached from the greedy solution $x'$ through a number of inserts and removals which maintain the knapsack filled close to the capacity. Using this concept, Pisinger [11] introduced a dynamic programming recursion with time and space complexity $O(nw_{\max}\Gamma)$ where $p_{\max} = \max_{j=1,\ldots,n} p_j$, $w_{\max} = \max_{j=1,\ldots,n} w_j$ and $\Gamma = z^* - z$ for any lower bound $z$. Since $\Gamma \leqslant p_{\max}$—as we may choose $z$ equal to the greedy solution—the complexity is bounded by the term $O(nw_{\max} p_{\max})$ which is linear in $n$ when the magnitude of the weights and profits is bounded by a constant. Note that the practical running times may be improved by finding a lower bound $z$ of good quality. If we are so lucky that the optimal solution has been found by some initial heuristic so that the balanced dynamic programming algorithm is used only for proving optimality, the time and space complexity is limited to $O(nw_{\max})$.

A final dynamic programming recursion was presented by Pisinger [12] in the framework of an *expanding core* algorithm. The Minknap algorithm is based on the ordinary Bellman recursion, but the recursion starts from the greedy solution and alternate between an insertion or a removal of an item. The running time is bounded by $O(n(b-a))$ where $\{a,\ldots,b\}$ is the set of items which needed to be enumerated. This set is also known as the "*minimal*" *core* of the problem.

The currently most successful algorithm for (KP) was presented by Martello et al. [13]. The algorithm can be seen as a combination of many different concepts and is hence called Combo. The enumeration part of Combo is based on the same dynamic programming algorithm as Minknap but additional techniques are gradually introduced if the problem seems to be hard to solve. A good indication of the hardness of the problem is the number of states in the dynamic programming algorithm—if it surpasses a given threshold value, we take it as an indication of need for additional techniques. Depending on the hardness of the problem, rudimentary divisibility techniques are used, followed by bounds from cardinality constraints, and finally followed by improved lower bounding through merging of items with the states in the dynamic programming. The three techniques are introduced gradually, and in most cases they result in a decrease of the states in the dynamic programming algorithm. In the worst case, we proceed with the ordinary Minknap recursion, but hopefully having tighter upper and lower bounds available.

## 3. Computational experiments

Despite the $\mathcal{NP}$-hardness of the knapsack problem, we strive towards developing algorithms which efficiently can solve a large variety of problems occurring in practice. Unfortunately, very few real-life instances of (KP) are reported in the literature, hence algorithm design has focused on a set of synthetic benchmark tests. These tests are all based on randomly distributed profits or weights, and hence implicitly contain some structure which may not be present in real-life instances.

In the following study, we will consider nearly all the knapsack algorithms presented in Section 1, including the MT2, MThard, Combo, Expknap, and Minknap algorithm. The dynamic programming algorithm based on balancing has never been implemented, although its worst-case complexity makes it a promising candidate. The word-RAM algorithm is very complicated and the constants hidden in the big-Oh notation are large. Hence, it is mainly interesting from a theoretical point of view, and no computational results are given. Concerning the performance of various older algorithms the comparison with MT2 reported in [14] should be consulted.

We will consider several groups of randomly generated instances of (KP) which have been constructed to reflect special properties that may influence the solution process. In all instances, the weights are uniformly distributed in a given interval with *data range* $R = 1000$ and $10\,000$. The profits are expressed as a function of the weights, yielding the specific properties of each group. The instance groups are graphically illustrated in Fig. 1.

- *Uncorrelated data instances*: $p_j$ and $w_j$ are chosen randomly in $[1, R]$. In these instances, there is no correlation between the profit and weight of an item. Such instances illustrate those situations where it is reasonable to assume that the profit does not depend on the weight. Uncorrelated instances are generally easy to solve, as there is a large variation between the profits and weights, making it easy to fathom numerous variables by upper bound tests or by dominance relations.
- *Weakly correlated instances*: Weights $w_j$ are chosen randomly in $[1, R]$ and the profits $p_j$ in $[w_j - R/10, w_j + R/10]$ such that $p_j \geqslant 1$. Despite their name, weakly correlated instances have a very high correlation between the profit and weight of an item. Typically, the profit differs from the weight by only a few percent. Such instances are perhaps the most realistic in management,

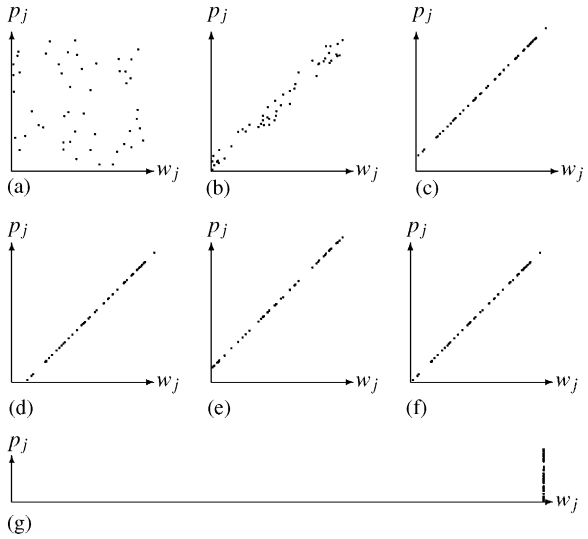


Fig. 1. Classical test instances: (a) uncorrelated instances, (b) weakly correlated instances, (c) strongly correlated instances, (d) inverse strongly correlated instances, (e) almost strongly correlated instances, (f) subset sum instances, (g) uncorrelated instances with similar weights. Note that instances (c) and (e) look very similar since the extra "noise" in almost strongly correlated instances is very small.

as it is well known that the return of an investment is generally proportional to the sum invested within some small variations.

- *Strongly correlated instances*: Weights $w_j$ are distributed in $[1,R]$ and $p_j = w_j + R/10$. Such instances correspond to a real-life situation where the return is proportional to the investment plus some fixed charge for each project. The strongly correlated instances are hard to solve for two reasons:
  (a) The instances are *ill-conditioned* in the sense that there is a large gap between the continuous and integer solution of the problem.
  (b) Sorting the items according to decreasing efficiencies correspond to a sorting according to the weights. Thus for any small interval of the ordered items (i.e. a "core") there is a limited variation in the weights, making it difficult to satisfy the capacity constraint with equality.
- *Inverse strongly correlated instances*: Profits $p_j$ are distributed in $[1,R]$ and $w_j = p_j + R/10$. These instances are like strongly correlated instances, but the fixed charge is negative.
- *Almost strongly correlated instances*: Weights $w_j$ are distributed in $[1,R]$ and the profits $p_j$ in $[w_j + R/10 - R/500, w_j + R/10 + R/500]$. These are a kind of fixed-charge problems with some noise added. Thus, they reflect the properties of both strongly and weakly correlated instances.
- *Subset sum instances*: Weights $w_j$ are randomly distributed in $[1,R]$ and $p_j = w_j$. These instances reflect the situation where the profit of each item is equal (or proportional) to the weight. Thus, our only goal is to obtain a filled knapsack. Subset sum instances are however challenging to solve as instances to (KP) because most of the considered upper bounds return the same trivial

Table 1
Average solution times in milliseconds, MT2 (Intel Pentium IV, 3 GHz)

| | Uncorr. | | Weak. corr. | | Str. corr. | | Inv. str. corr. | | Al. str. corr. | | Subs. sum | | Sim. $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.1 | 1.8 | 1.3 | 0.3 | 0.6 | 0.9 | 0.9 | 0.0 | 0.1 | 0.5 |
| 100 | 0.0 | 0.1 | 0.0 | 0.0 | 663.1 | 671.7 | 106.4 | 4566.8 | 170.5 | 484.5 | 0.1 | 0.3 | 96.2 |
| 200 | 0.0 | 0.0 | 0.0 | 0.1 | — | — | — | — | — | — | 0.0 | 0.8 | — |
| 500 | 0.1 | 0.1 | 0.1 | 0.3 | — | — | — | — | — | — | 0.1 | 0.5 | — |
| 1000 | 0.2 | 0.2 | 0.3 | 0.6 | — | — | — | — | — | — | 0.1 | 0.5 | — |
| 2000 | 0.3 | 0.4 | 0.4 | 1.0 | — | — | — | — | — | — | 0.1 | 0.4 | — |
| 5000 | 0.5 | 0.7 | 0.6 | 2.3 | — | — | — | — | — | — | 0.3 | 0.8 | — |
| 10 000 | 0.9 | 1.4 | 0.8 | 3.1 | — | — | — | — | — | — | 0.2 | 0.7 | — |

value $c$, and thus we cannot use bounding rules for cutting off branches before an optimal solution has been found.

- *Uncorrelated instances with similar weights*: Weights $w_j$ are distributed in $[100\,000, 100\,100]$ and the profits $p_j$ in $[1, 1000]$.

We will compare the performance of MT2, Expknap, Minknap, MThard and Combo. The behavior of the algorithms will be considered for different problem sizes $n$, different problem types, and two data ranges. For each instance type a series of $H = 100$ instances is performed. The capacity in each instance is chosen as

$$c = \frac{h}{H+1} \sum_{j=1}^{n} w_j \tag{5}$$

for test instance number $h = 1, \ldots, H$. This is done to "smooth out" variations due to the choice of capacity as described in [15].

All tests were run on an Intel Pentium IV, 3 GHz with 1 Gb RAM, and a time limit of 60 min was assigned to each instance type for all $H$ instances. If not all instances were solved within the time or space limit, this is indicated by a dash in the table.

Tables 1–5 compare the solution times of the five algorithms. The simple branch-and-bound codes MT2 and Expknap, have the overall worst performance, although they are quite fast on easy instances like the uncorrelated and weakly correlated ones. Moreover, they are among the fastest codes for the subset sum instances. For the different variants of strongly correlated instances, MT2 and Expknap are able to solve only tiny instances.

The dynamic programming algorithm Minknap has an overall stable performance, as it is able to solve all instances within reasonable time. It is the fastest code for uncorrelated and weakly correlated instances and it has almost the same good performance for subset sum instances as the branch-and-bound algorithms. The strongly correlated instances take considerably more time to be solved but although Minknap uses only simple bounds from LP-relaxation, the pseudo-polynomial time complexity gets it safely through these instances.

Table 2
Average solution times in milliseconds, Expknap (Intel Pentium IV, 3 GHz)

| | Uncorr. | | Weak. corr. | | Str. corr. | | Inv. str. corr. | | Al. str. corr. | | Subs. sum | | Sim. $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.0 | 4.0 | 1.6 | 0.6 | 1.7 | 1.6 | 1.3 | 0.0 | 0.2 | 2.3 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 | 3903.9 | 1981.0 | 704.4 | 11 387.0 | 357.9 | 682.1 | 0.0 | 0.3 | 871.2 |
| 200 | 0.0 | 0.0 | 0.1 | 0.1 | — | — | — | — | — | — | 0.0 | 0.3 | — |
| 500 | 0.0 | 0.0 | 0.2 | 0.4 | — | — | — | — | — | — | 0.0 | 0.4 | — |
| 1000 | 0.0 | 0.1 | 0.2 | 0.7 | — | — | — | — | — | — | 0.0 | 0.3 | — |
| 2000 | 0.1 | 0.2 | 0.1 | 2.0 | — | — | — | — | — | — | 0.0 | 0.5 | — |
| 5000 | 0.3 | 0.6 | 0.2 | 5.7 | — | — | — | — | — | — | 0.1 | 0.7 | — |
| 10 000 | 0.8 | 2.1 | 0.4 | 9.5 | — | — | — | — | — | — | 0.3 | 0.7 | — |

Table 3
Average solution times in milliseconds, Minknap (Intel Pentium IV, 3 GHz)

| | Uncorr. | | Weak. corr. | | Str. corr. | | Inv. str. corr. | | Al. str. corr. | | Subs. sum | | Sim. $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.2 | 1.0 | 0.2 | 0.6 | 0.2 | 0.4 | 0.1 | 1.3 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 | 0.8 | 5.6 | 0.8 | 4.5 | 0.5 | 1.3 | 0.1 | 1.3 | 0.3 |
| 200 | 0.0 | 0.0 | 0.1 | 0.2 | 2.4 | 23.9 | 1.9 | 16.1 | 2.2 | 6.1 | 0.0 | 1.4 | 0.2 |
| 500 | 0.1 | 0.1 | 0.2 | 0.1 | 7.4 | 70.5 | 7.7 | 64.8 | 6.2 | 33.3 | 0.2 | 1.1 | 1.5 |
| 1000 | 0.1 | 0.2 | 0.1 | 0.2 | 17.7 | 222.2 | 15.6 | 184.6 | 14.0 | 111.0 | 0.1 | 1.3 | 5.4 |
| 2000 | 0.3 | 0.1 | 0.1 | 0.3 | 34.1 | 353.0 | 37.8 | 379.4 | 26.4 | 277.3 | 0.0 | 1.5 | 17.6 |
| 5000 | 0.3 | 0.4 | 0.1 | 1.0 | 123.5 | 1368.2 | 105.6 | 1433.2 | 55.4 | 797.1 | 0.3 | 2.0 | 59.2 |
| 10 000 | 0.5 | 0.4 | 0.5 | 2.0 | 261.8 | 2985.0 | 204.8 | 3096.5 | 62.3 | 1312.1 | 0.2 | 1.7 | 69.8 |

Table 4
Average solution times in milliseconds, MThard (Intel Pentium IV, 3 GHz)

| | Uncorr. | | Weak. corr. | | Str. corr. | | Inv. str. corr. | | Al. str. corr. | | Subs. sum | | Sim. $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.2 | 0.3 | 0.2 | 0.1 | 0.4 | 0.8 | 0.0 | 0.2 | 0.2 |
| 100 | 0.0 | 0.0 | 0.0 | 0.1 | 0.5 | 0.6 | 0.2 | 0.4 | 1.0 | 3.9 | 0.1 | 0.2 | 0.4 |
| 200 | 0.1 | 0.1 | 0.1 | 0.1 | 1.7 | 2.0 | 0.8 | 1.1 | 2.7 | 12.9 | 0.1 | 0.6 | 1.1 |
| 500 | 0.2 | 0.2 | 0.2 | 0.2 | 4.0 | 4.3 | 2.3 | 2.4 | 4.1 | 34.7 | 0.1 | 0.5 | 2.5 |
| 1000 | 0.3 | 0.3 | 0.5 | 0.8 | 7.3 | 11.6 | 4.1 | 5.2 | 7.6 | 49.0 | 0.2 | 0.8 | 4.5 |
| 2000 | 0.3 | 0.7 | 0.6 | 1.3 | 8.9 | 20.2 | 5.0 | 8.5 | 12.8 | 41.1 | 0.0 | 0.6 | 7.5 |
| 5000 | 1.1 | 1.3 | 0.6 | 2.4 | 9.7 | 103.0 | 9.6 | 27.2 | 126.1 | 59.6 | 0.1 | 0.5 | 10.0 |
| 10 000 | 1.6 | 2.9 | 1.0 | 4.0 | 14.8 | 281.2 | 12.9 | 67.9 | 347 424.7 | 21 298.3 | 0.5 | 32.2 | 14.2 |

Table 5
Average solution times in milliseconds, Combo (Intel Pentium IV, 3 GHz)

| | Uncorr. | | Weak. corr. | | Str. corr. | | Inv. str. corr. | | Al. str. corr. | | Subs. sum | | Sim. $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^3$ | $10^4$ | $10^5$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.1 | 0.2 | 0.3 | 0.1 | 0.4 | 0.1 | 0.2 | 0.0 | 0.7 | 0.1 |
| 100 | 0.0 | 0.0 | 0.0 | 0.1 | 0.4 | 0.7 | 0.3 | 0.9 | 0.6 | 0.7 | 0.0 | 0.6 | 0.0 |
| 200 | 0.0 | 0.0 | 0.1 | 0.0 | 0.6 | 0.8 | 0.4 | 1.3 | 0.4 | 1.2 | 0.2 | 0.6 | 0.3 |
| 500 | 0.1 | 0.1 | 0.0 | 0.2 | 0.7 | 1.7 | 0.4 | 1.1 | 0.9 | 0.8 | 0.1 | 0.9 | 1.0 |
| 1000 | 0.2 | 0.0 | 0.2 | 0.2 | 0.6 | 1.5 | 1.0 | 1.5 | 1.0 | 0.7 | 0.1 | 0.7 | 2.1 |
| 2000 | 0.1 | 0.4 | 0.2 | 0.6 | 1.0 | 1.1 | 1.1 | 1.6 | 0.8 | 1.2 | 0.0 | 0.8 | 2.5 |
| 5000 | 0.5 | 0.4 | 0.2 | 1.2 | 1.7 | 1.7 | 1.4 | 2.1 | 0.9 | 2.1 | 0.1 | 1.1 | 4.2 |
| 10 000 | 0.9 | 1.1 | 0.8 | 1.9 | 2.9 | 3.2 | 2.6 | 3.9 | 2.6 | 2.7 | 0.2 | 0.6 | 4.9 |

The MThard algorithm has a very good overall performance, as it is able to solve nearly all problems within split seconds. The short solution times are mainly due to its cardinality bounds which make it possible to terminate the branching after having explored a small number of nodes. There are, however, some anomalous entries for large-sized almost strongly correlated problems, where the cardinality bounds somehow fail. Also for some large-sized subset sum problems MThard takes unproportionally long time.

The best performance is obtained with the Combo algorithm. This "hybrid" algorithm solves all the considered instances within 5 ms on average and the running times have a very systematic growth rate with small variations. In particular, it is worth noting that the ratio between solving a difficult instance and an easy instance is within a factor of 10, thus showing that the additional work needed to derive tight bounds can be done very fast.

Based on the results in Table 5 one could draw the wrong conclusion that (KP) is easy to solve. One should, however, notice that we consider instances where all the coefficients are of moderate size. If a real-life instance has this property (or can be scaled down to satisfy the property without significantly affecting the solution), then the problems are indeed easy to solve. However, there still exist many applications where more or less intractable knapsack instances with very large coefficients occur.

### 3.1. Difficult instances

There are two directions to follow when constructing difficult instances: one may either consider the traditional instances with larger coefficients. This will obviously make the dynamic programming algorithms run slower, but also the upper bounds get weakened since the gap to the optimal solution is scaled and cannot be closed by simply rounding down the upper bound to the nearest smaller integer. A second direction to follow is to construct instances where the coefficients are of moderate size, but where all currently used upper bounds have a bad performance.

### 3.2. Difficult instances with large coefficients

Our first attempt to construct difficult instances is to use the "standard" instances from Section 3 for increasing data range $R$. The motivation being, that we know that the magnitude of the weights

Table 6
Average solution times in milliseconds, Expknap, for large range instances (Intel Pentium IV, 3 GHz)

| | Uncorr. | | | Weak. corr. | | | Str. corr. | | | Inv. str. corr. | | | Al. str. corr. | | | Subs. sum | | | Sim. $w$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^7$ | $10^8$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 6.2 | 3.1 | 4.8 | 3.1 | 2.6 | 1.7 | 3.4 | 1.3 | 1.1 | 2.9 | 22.9 | 224.1 | 1.3 | 1.4 |
| 100 | 0.1 | 0.1 | 0.0 | 0.1 | 0.1 | 0.1 | 5611.7 | 1391.6 | 3415.2 | 1340.4 | 693.6 | 2032.9 | 651.0 | 1725.3 | 546.5 | 3.2 | 23.2 | 351.6 | 1253.6 | 1048.5 |
| 200 | 0.0 | 0.0 | 0.0 | 0.2 | 0.1 | 0.2 | — | — | — | — | — | — | — | — | — | 3.3 | 39.0 | 302.8 | — | — |
| 500 | 0.1 | 0.1 | 0.0 | 0.7 | 0.6 | 0.6 | — | — | — | — | — | — | — | — | — | 3.1 | 33.9 | 354.5 | — | — |
| 1000 | 0.2 | 0.2 | 0.1 | 1.6 | 1.8 | 2.1 | — | — | — | — | — | — | — | — | — | 3.3 | 30.3 | 385.7 | — | — |
| 2000 | 0.2 | 0.4 | 0.2 | 4.7 | 5.3 | 5.6 | — | — | — | — | — | — | — | — | — | 3.8 | 30.3 | 299.3 | — | — |
| 5000 | 1.2 | 1.6 | 1.4 | 16.9 | 23.2 | 18.3 | — | — | — | — | — | — | — | — | — | 3.7 | 33.1 | 268.7 | — | — |
| 10 000 | 3.7 | 4.0 | 3.8 | 53.4 | 51.7 | 44.2 | — | — | — | — | — | — | — | — | — | 4.3 | 34.8 | 415.3 | — | — |

Table 7
Average solution times in milliseconds, Minknap, for large range instances (Intel Pentium IV, 3 GHz)

| | Uncorr. | | | Weak. corr. | | | Str. corr. | | | Inv. str. corr. | | | Al. str. corr. | | | Subs. sum | | | Sim. $w$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^7$ | $10^8$ |
| 50 | 0.0 | 0.0 | 0.2 | 0.0 | 0.1 | 0.1 | 8.3 | 8.9 | 13.6 | 3.6 | 7.4 | 5.1 | 0.9 | 0.8 | 0.7 | 29.1 | 236.4 | — | 0.0 | 0.0 |
| 100 | 0.0 | 0.1 | 0.0 | 0.1 | 0.2 | 0.0 | 82.3 | 408.1 | — | 59.7 | 271.6 | 1190.1 | 4.7 | 5.0 | 4.1 | 29.1 | 309.4 | — | 0.2 | 0.0 |
| 200 | 0.0 | 0.1 | 0.0 | 0.3 | 0.2 | 0.3 | 436.8 | 3276.7 | — | 437.1 | 3291.1 | — | 21.5 | 25.2 | 21.5 | 32.3 | 388.0 | — | 0.7 | 0.7 |
| 500 | 0.2 | 0.1 | 0.0 | 0.0 | 0.5 | 0.4 | 1698.0 | — | — | 1713.1 | — | — | 129.1 | 150.5 | 130.1 | 34.0 | 351.3 | — | 3.8 | 3.8 |
| 1000 | 0.3 | 0.1 | 0.5 | 0.6 | 0.7 | 0.9 | — | — | — | 4864.5 | — | — | 365.5 | 458.7 | 471.2 | 29.3 | 420.8 | — | 14.4 | 15.0 |
| 2000 | 0.6 | 0.1 | 0.2 | 1.5 | 1.9 | 1.8 | — | — | — | 10 204.1 | — | — | 1152.9 | 1456.2 | 1381.6 | 32.6 | 407.4 | — | 68.1 | 62.2 |
| 5000 | 0.8 | 1.3 | 0.9 | 4.0 | 4.0 | 4.1 | — | — | — | 29 366.2 | — | — | 4090.0 | 5616.1 | 6035.9 | 35.5 | 378.9 | — | 349.0 | 358.5 |
| 10 000 | 1.9 | 1.8 | 1.4 | 7.2 | 8.2 | 8.1 | — | — | — | — | — | — | 8377.6 | 16 511.9 | 14 123.5 | 41.6 | 418.1 | — | 809.7 | 797.5 |

strictly affect the computational complexity of dynamic programming algorithms; hence, in this way the effect of upper bound tests, reduction, and early termination becomes more clear.

For the following experiments, the codes were compiled using 64-bit integers, making it possible to run tests with weights of considerable size. The dynamic programming tables of Minknap and Combo were extended significantly, and the subgradient algorithm used in Combo for finding appropriate surrogate multipliers in the bounding procedure was slightly modified. For technical reasons, it was not possible to modify the FORTRAN codes MT2 and MThard to 64 bit integers.

The outcome of the experiments is shown in Tables 6–8. Note that the increased integer size implies a slow-down by a factor of at least 2 compared to the tests with small data range. Apart from this observation, it is interesting to see that uncorrelated and weakly correlated instances are almost unaffected by an increase of the data range. However, for nearly all other instance types the problems become harder as the data range is increased. Several instances cannot be solved within the given time or space limit, and it is also interesting to note that the upper bounds used in Combo for some instances have difficulties in closing the gap between the upper and lower bound. The dynamic programming part of Minknap and Combo ensures that the optimal solution is found in pseudo-polynomial time, but as the data range is increased this time bound starts to grow unacceptably high. For data range $R = 10^7$, Minknap and Combo frequently run out of space before running out of time.

The main observations from the computational experiments with small-range instances are, however, still valid. First of all, we notice that the methods used for finding tighter upper and lower

Table 8
Average solution times in milliseconds, Combo, for large range instances (Intel Pentium IV, 3 GHz)

| | Uncorr. | | | Weak. corr. | | | Str. corr. | | | Inv. str. corr. | | | Al. str. corr. | | | Subs. sum | | | Sim. $w$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ \ $R$: | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^5$ | $10^6$ | $10^7$ | $10^7$ | $10^8$ |
| 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.2 | 0.0 | 22.0 | 248.7 | — | 4.2 | 182.4 | — | 1.4 | 1.2 | 1.0 | 20.9 | 295.5 | — | 0.1 | 0.2 |
| 100 | 0.0 | 0.0 | 0.0 | 0.3 | 0.1 | 0.0 | 11.1 | 348.7 | — | 7.0 | 284.8 | — | 3.4 | 3.8 | 3.6 | 12.6 | 313.4 | — | 0.3 | 0.3 |
| 200 | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 0.3 | 18.0 | 421.0 | — | 11.8 | 334.4 | — | 6.4 | 5.6 | 6.4 | 8.5 | 179.9 | — | 0.8 | 1.0 |
| 500 | 0.2 | 0.1 | 0.0 | 0.4 | 0.6 | 0.7 | 10.7 | 321.5 | — | 11.9 | 278.1 | — | 5.8 | 5.7 | 6.3 | 8.4 | 106.4 | — | 5.4 | 4.9 |
| 1000 | 0.2 | 0.1 | 0.1 | 1.0 | 1.3 | 1.2 | 12.0 | 107.5 | — | 12.1 | 39.4 | — | 7.2 | 6.2 | 6.1 | 7.2 | 73.8 | — | 6.9 | 8.3 |
| 2000 | 0.5 | 0.6 | 0.3 | 2.5 | 2.8 | 2.3 | 8.8 | 46.3 | — | 11.8 | 41.4 | — | 5.0 | 8.4 | 7.4 | 7.2 | 30.6 | — | 6.5 | 6.9 |
| 5000 | 0.8 | 1.3 | 0.7 | 6.2 | 6.8 | 6.2 | 8.6 | 19.6 | — | 12.9 | 15.3 | — | 6.9 | 8.7 | 9.5 | 7.5 | 10.5 | — | 7.7 | 7.6 |
| 10 000 | 2.6 | 1.9 | 1.9 | 12.1 | 13.0 | 11.9 | 13.2 | 21.9 | — | 13.4 | 19.6 | — | 8.3 | 11.9 | 12.9 | 7.8 | 11.6 | — | 8.2 | 8.9 |

bounds in Combo do pay off, since in general Combo is able to solve considerably more instances than the "clean" dynamic programming version Minknap. It is interesting to observe that some of the strongly correlated problems tend to become easier for Combo when $n$ is increasing. Next, we notice that branch-and-bound methods, in general, cannot compete with dynamic programming methods. There is actually a single exception to this observation for the subset sum instances. The randomly generated subset sum instances have numerous solution to the decision problem and hence a branch-and-bound algorithm may terminate the search as soon as an optimal solution has been found.

### 3.3. Difficult instances with small coefficients

It is more challenging to construct instances with small coefficients, where present algorithms perform badly. Obviously, the worst-case complexity of dynamic programming algorithms still holds, but one may construct the instances so that it is difficult to fathom states through upper bound tests when using dynamic programming. The following classes of instances have been identified as having the desired property (Fig. 2).

- *Spanner instances* span$(v,m)$: These instances are constructed such that all items are multiples of a quite small set of items—the so-called spanner set. The spanner instances span$(v,m)$ are characterized by the following three parameters: $v$ is the size of the spanner set, $m$ is the multiplier limit, and finally we may have any distribution (uncorrelated, weakly correlated, strongly correlated, etc.) of the items in the spanner set.
  More formally, the instances are generated as follows: A set of $v$ items is generated with weights in the interval $[1,R]$, and profits according to the distribution. The items $(p_k,w_k)$ in the spanner set are normalized by setting $p_k := \lceil 2p_k/m \rceil$ and $w_k := \lceil 2w_k/m \rceil$. The $n$ items are then constructed, by repeatedly choosing an item $(p_k,w_k)$ from the spanner set, and a multiplier $a$ randomly generated in the interval $[1,m]$. The constructed item has profit and weight $(a \cdot p_k, a \cdot w_k)$.
  Three distributions of the spanner problems span$(s,m)$ have been considered: uncorrelated, weakly correlated, and strongly correlated. The multiplier limit was chosen as $m = 10$. Computational

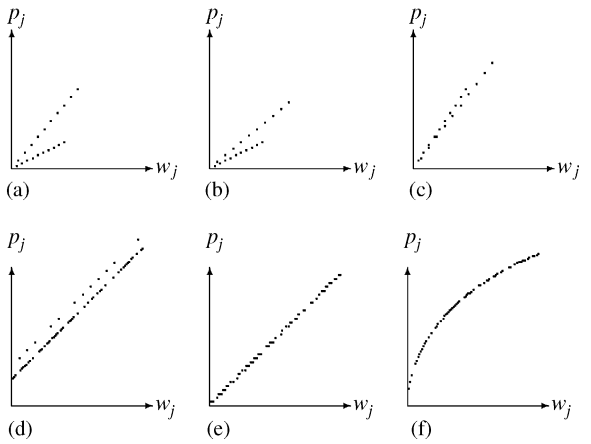


Fig. 2. New test instances considered: (a) uncorrelated spanner instances $\mathsf{span}(2,10)$, (b) weakly correlated spanner instances $\mathsf{span}(2,10)$, (c) strongly correlated spanner instances $\mathsf{span}(2,10)$, (d) multiple strongly correlated instances $\mathsf{mstr}(3R/10,2R/10,6)$, (e) profit ceiling instances $\mathsf{pceil}(3)$, (f) circle instances $\mathsf{circle}(\frac{2}{3})$.

experiments showed that the instances became harder to solve for smaller spanner sets. Hence, in the following we will consider the instances: uncorrelated $\mathsf{span}(2,10)$, weakly correlated $\mathsf{span}(2,10)$, and strongly correlated $\mathsf{span}(2,10)$.

- *Multiple strongly correlated instances* $\mathsf{mstr}(k_1,k_2,d)$: These instances are constructed as a combination of two sets of strongly correlated instances. Both instances have profits $p_j := w_j + k_i$ where $k_i$, $i = 1,2$ is different for the two instances.
  The multiple strongly correlated instances $\mathsf{mstr}(k_1,k_2,d)$ are generated as follows: the weights of the $n$ items are randomly distributed in $[1,R]$. If the weight $w_j$ is divisible by $d$, then we set the profit $p_j := w_j + k_1$ otherwise set it to $p_j := w_j + k_2$. Notice that the weights $w_j$ in the first group (i.e. where $p_j = w_j + k_1$) will all be multiples of $d$, so that using only these weights we can at most use $d\lfloor c/d \rfloor$ of the capacity. To obtain a completely filled knapsack, we need to include some of the items from the second group.
  Computational experiments showed that very difficult instances could be obtained with the parameters $\mathsf{mstr}(3R/10,2R/10,d)$. Choosing $d = 6$ results in the most difficult instances, but values of $d$ between 3 and 10 can all be used.
- *Profit ceiling instances* $\mathsf{pceil}(d)$: These instances have the property that all profits are multiples of a given parameter $d$. The weights of the $n$ items are randomly distributed in $[1,R]$, and the profits are set to $p_j = d\lceil w_j/d \rceil$.
  The parameter $d$ was experimentally chosen as $d = 3$, as this resulted in sufficiently difficult instances.
- *Circle instances* $\mathsf{circle}(d)$: The instances $\mathsf{circle}(d)$ are generated such that the profits as function of the weights form an arc of a circle (actually an ellipsis). The weights are uniformly distributed in $[1,R]$ and for each weight $w$ the corresponding profit is chosen as $p = d\sqrt{4R^2 - (w-2R)^2}$.
  The computational study showed that difficult instances appeared by choosing $d = \frac{2}{3}$ which was chosen for the following experiments.

Table 9
Solution times in milliseconds, MT2, for difficult instances with small coefficients (Intel Pentium IV, 3 GHz)

| $n$ | Uncorr. span$(2,10)$ | Weak. corr. span$(2,10)$ | Str. corr. span$(2,10)$ | mstr$(3R/10,2R/10,6)$ | pceil$(3)$ | circle$(\frac{2}{3})$ |
|---|---|---|---|---|---|---|
| 20 | 0.0 | 0.3 | 0.2 | 0.0 | 0.0 | 0.0 |
| 50 | 3112.0 | — | — | 0.2 | 4.6 | 0.1 |
| 100 | — | — | — | 24.7 | — | 6.2 |
| 200 | — | — | — | — | — | — |
| 500 | — | — | — | — | — | — |
| 1000 | — | — | — | — | — | — |
| 2000 | — | — | — | — | — | — |
| 5000 | — | — | — | — | — | — |
| 10 000 | — | — | — | — | — | — |

Table 10
Solution times in milliseconds, Expknap, for difficult instances with small coefficients (Intel Pentium IV, 3 GHz)

| $n$ | Uncorr. span$(2,10)$ | Weak. corr. span$(2,10)$ | Str. corr. span$(2,10)$ | mstr$(3R/10,2R/10,6)$ | pceil$(3)$ | circle$(\frac{2}{3})$ |
|---|---|---|---|---|---|---|
| 20 | 0.0 | 0.2 | 0.1 | 0.0 | 0.1 | 0.0 |
| 50 | 934.9 | — | — | 0.4 | 6.5 | 0.1 |
| 100 | — | — | — | 21.4 | — | 15.4 |
| 200 | — | — | — | — | — | 9280.9 |
| 500 | — | — | — | — | — | — |
| 1000 | — | — | — | — | — | — |
| 2000 | — | — | — | — | — | — |
| 5000 | — | — | — | — | — | — |
| 10 000 | — | — | — | — | — | — |

For each instance type a series of $H = 100$ instances is performed, and the capacity is chosen as in (5). Since all the above "difficult" instances are generated with moderate data range $R = 1000$, all the codes MT2, Expknap, Minknap, MThard and Combo could be used in the normal version using 32 bit integers. The outcome of the experiments is shown in Tables 9–13.

It appears, that the branch-and-bound algorithms MT2 and Expknap are able to solve this kind of instances only for small values of $n$. The MThard algorithm, which is a combination of branch-and-bound and dynamic programming has a slightly better performance, being able to solve instances of moderate size. Both of the two dynamic programming algorithms Minknap and Combo are able to solve also large-sized instances since the worst-case running time of $O(nc)$ is acceptable for small values of the data range. It is, however, interesting to observe that the tighter bounds in Combo do not significantly improve the running times. Indeed, Minknap is able to solve some of the strongly correlated spanner instances faster than Combo using only dynamic programming and computationally cheap upper bounds.

Table 11
Solution times in milliseconds, Minknap, for difficult instances with small coefficients (Intel Pentium IV, 3 GHz)

| $n$ | Uncorr. span(2, 10) | Weak. corr. span(2, 10) | Str. corr. span(2, 10) | mstr($3R/10, 2R/10, 6$) | pceil(3) | circle($\frac{2}{3}$) |
|---|---|---|---|---|---|---|
| 20 | 0.0 | 0.0 | 0.0 | 0.1 | 0.1 | 0.0 |
| 50 | 0.0 | 0.2 | 0.0 | 0.0 | 0.2 | 0.1 |
| 100 | 0.2 | 0.2 | 0.2 | 0.5 | 0.7 | 0.4 |
| 200 | 0.5 | 1.0 | 0.6 | 1.6 | 1.8 | 1.2 |
| 500 | 2.2 | 4.9 | 5.0 | 4.6 | 9.7 | 5.1 |
| 1000 | 9.0 | 17.7 | 15.3 | 9.8 | 25.5 | 10.4 |
| 2000 | 36.3 | 71.5 | 73.1 | 20.4 | 94.9 | 22.7 |
| 5000 | 205.9 | 419.4 | 448.0 | 71.8 | 744.0 | 76.6 |
| 10 000 | 899.5 | 1705.6 | 2207.6 | 177.2 | 2794.7 | 131.3 |

Table 12
Solution times in milliseconds, MThard, for difficult instances with small coefficients

| $n$ | Uncorr. span(2, 10) | Weak. corr. span(2, 10) | Str. corr. span(2, 10) | mstr($3R/10, 2R/10, 6$) | pceil(3) | circle($\frac{2}{3}$) |
|---|---|---|---|---|---|---|
| 20 | 0.0 | 0.1 | 0.2 | 0.0 | 0.1 | 0.0 |
| 50 | 1.4 | 1.9 | 2.1 | 0.1 | 0.7 | 0.1 |
| 100 | 5.6 | 10.1 | 8.0 | 0.5 | 6.1 | 0.9 |
| 200 | 2039.5 | 8806.1 | 4221.5 | 4.9 | 10 559.2 | 4.7 |
| 500 | — | — | — | 7228.6 | — | 29.2 |
| 1000 | — | — | — | — | — | 172.4 |
| 2000 | — | — | — | — | — | 1028.7 |
| 5000 | — | — | — | — | — | — |
| 10 000 | — | — | — | — | — | — |

Table 13
Solution times in milliseconds, Combo, for difficult instances with small coefficients

| $n$ | Uncorr. span(2, 10) | Weak. corr. span(2, 10) | Str. corr. span(2, 10) | mstr($3R/10, 2R/10, 6$) | pceil(3) | circle($\frac{2}{3}$) |
|---|---|---|---|---|---|---|
| 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 50 | 0.0 | 0.0 | 0.3 | 0.1 | 0.1 | 0.1 |
| 100 | 0.1 | 0.1 | 0.3 | 0.3 | 0.5 | 0.2 |
| 200 | 0.4 | 0.8 | 0.7 | 0.8 | 0.8 | 0.5 |
| 500 | 1.9 | 3.7 | 4.0 | 1.8 | 6.9 | 1.9 |
| 1000 | 6.5 | 12.7 | 13.6 | 4.2 | 19.4 | 6.1 |
| 2000 | 24.0 | 52.4 | 73.5 | 10.2 | 72.0 | 13.0 |
| 5000 | 134.9 | 392.6 | 434.9 | 38.7 | 652.5 | 56.4 |
| 10 000 | 629.3 | 1523.4 | 2272.6 | 97.3 | 2434.3 | 105.7 |

## 4. Conclusion

We have compared the solution times of all recent algorithms, using classical and new benchmark tests. The classical instances are generally easy to solve and hence one could wrongly conclude that no more research is needed for the knapsack problem. The new classes of benchmark tests clearly show that this is not the case. There are numerous interesting instances for which all currently known upper bounds perform badly, and for which the running times of the algorithms are close to the worst-case time-bound.

Dynamic programming is one of our best approaches for solving difficult (KP), since this is the only solution method which gives us a worst-case guarantee on the running time, independently on whether the upper bounding tests will work well.

Research in upper bounds working on more general classes of problems should also be stimulated. The knapsack polytope is one of the best studied (see e.g. Weismantel [16]) and numerous valid inequalities have been found which can be used to strengthen the LP-formulation of the (KP). Although a branch-and-cut framework may seem as an over-kill for knapsack problems, such techniques may become necessary in order to solve the more difficult instances.

## References

[1] Martello S, Pisinger D, Toth P. New trends in exact algorithms for the 0–1 knapsack problem. European Journal of Operational Research 2000;123:325–32.
[2] Kellerer H, Pferschy U, Pisinger D. Knapsack problems. Berlin: Springer; 2004.
[3] Meyer auf der Heide F. A polynomial linear search algorithm for the *n*-dimensional knapsack problem. Journal of the ACM 1984;31:668–76.
[4] Fournier H, Koiran P. Are lower bounds easier over the reals? In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC), 1998. p. 507–13.
[5] Balas E, Zemel E. An algorithm for large zero-one knapsack problems. Operations Research 1980;28:1130–54.
[6] Martello S, Toth P. A new algorithm for the 0–1 knapsack problem. Management Science 1988;34:633–44.
[7] Pisinger D. An expanding-core algorithm for the exact 0–1 knapsack problem. European Journal of Operational Research 1995;87:175–87.
[8] Martello S, Toth P. Upper bounds and algorithms for hard 0–1 knapsack problems. Operations Research 1997;45:768–78.
[9] Bellman RE. Dynamic programming. Princeton, NJ: Princeton University Press; 1957.
[10] Pisinger D. Dynamic programming on the word RAM. Algorithmica 2003;35:128–45.
[11] Pisinger D. Linear time algorithms for knapsack problems with bounded weights. Journal of Algorithms 1999;33:1–14.
[12] Pisinger D. A minimal algorithm for the 0–1 knapsack problem. Operations Research 1997;45:758–67.
[13] Martello S, Pisinger D, Toth P. Dynamic programming and strong bounds for the 0–1 knapsack problem. Management Science 1999;45:414–24.
[14] Martello S, Toth P. Knapsack problems: algorithms and computer implementations. New York: Wiley; 1990.
[15] Pisinger D. Core problems in knapsack algorithms. Operations Research 1999;47:570–5.
[16] Weismantel R. On the 0/1 knapsack polytope. Mathematical Programming 1997;77:49–68.