A Minimal Algorithm for the Bounded Knapsack Problem.*

David Pisinger

Dept. of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen, Denmark.

September, 1994

Abstract

The Bounded Knapsack Problem (BKP) is a generalization of the 0-1 Knapsack Problem where a bounded amount of each item type is available. The currently most efficient algorithm for BKP transforms the data instance to an equivalent 0-1 Knapsack Problem, which is solved efficiently through a specialized algorithm. However this paper demonstrates, that the transformation introduces many similar weighted items, resulting in very hard instances of the 0-1 Knapsack Problem.

To avoid these problems, a specialized algorithm is proposed which solves an expanding core problem through dynamic programming, such that the number of enumerated item types is minimal. Sorting and reduction is done by need, resulting in very little effort for the preprocessing. Compared to other algorithms for BKP, the presented algorithm uses tighter reductions and enumerates considerably less item types. Computational experiments are presented, showing that the presented algorithm outperforms all previously published algorithms for BKP.

Keywords: Packing; Bounded Knapsack Problem; Dynamic Programming; Reduction;

1 Introduction

Given n *item types* to pack in some knapsack of *capacity* c. Each item type j has a *profit* p_j, *weight* w_j, and a *bound* m_j on the availability. The problem is to select a number x_j $(0 \leq x_j \leq m_j)$ of each item type such that the profit sum of the included items is maximized without having the weight sum to exceed c. The *Bounded Knapsack Problem* (*BKP*) may thus be defined as the following optimization problem:

$$\max \qquad z = \sum_{j=1}^{n} p_j x_j$$

*Technical Report 94/27, DIKU, University of Copenhagen, Denmark.

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le c, \tag{1}$$
$$x_j \in \{0, 1, \ldots, m_j\}, \quad j = 1, \ldots, n,$$

where all coefficients are positive integers. Without loss of generality we may assume that $m_j w_j \le c$ for $j = 1, \ldots, n$ so all items avilable of a given type fits into the knapsack, and that $\sum_{j=1}^{n} m_j w_j > c$ to ensure a nontrivial problem. If we relax the integrality constraint $x_j \in \{0, 1, \ldots, m_j\}$ in (1) to the linear constraint $0 \le x_j \le m_j$, we obtain the *Linear Bounded Knapsack Problem* (*LBKP*). If $m_j = 1$ for all item types we get the well-known *0-1 Knapsack Problem* (*KP*).

BKP is NP-hard as it contains KP as a special case, but it can be solved in pseudo-polynomial time through dynamic programming (Martello and Toth 1990a). The present paper is devoted to data instances with moderate values of m_j, since if the bound on each item type is wide, other kinds of algorithms perform better — especially for the unbounded knapsack problem where $m_j = \infty$, $j = 1, \ldots, n$ (see Martello and Toth 1990b).

Several industrial problems which usually are solved as 0-1 Knapsack Problems may equally well be formulated as Bounded Knapsack Problems, thus taking advantage of the fact, that most products come from series of identical item types. Many combinatorial problems can be reduced to BKP, and the problem arises also as a subproblem in several algorithms of integer linear programming.

Only a few specialized algorithms for BKP have been published, but the following should be mentioned: Martello and Toth (1977), Ingargiola and Korsh (1977), Bulfin, Parker and Shetty (1979). However Martello and Toth (1990a) demonstrate, that all these algorithms are outperformed by their MTB2 algorithm, that transforms the BKP to an equivalent KP which is solved by a specialized algorithm. The transformation from BKP to KP is based on a binary coding of the bound m_j on each item type j. Thus each item type is replaced by $\lfloor \log_2 m_j \rfloor + 1$ items in the KP case, as follows:

$$(p_j, w_j),\ (2p_j, 2w_j),\ (4p_j, 4w_j), \ldots, (2^{a-1}p_j, 2^{a-1}w_j),\ (dp_j, dw_j), \tag{2}$$

where $a = \lfloor \log_2 m_j \rfloor$, and $d = m_j - \sum_{i=0}^{a-1} 2^i$. This approach is very elegant, but several problems arise:

- Many new variables are introduced by the transformation, implying that Martello and Toth (1990a) are not able to solve very large data instances ($n > 50.000$) due to memory limitations.

- Pisinger (1994e) showed that 0-1 Knapsack Problems with many similarly weighted items are hard to solve, since it is difficult to combine the items such that a filled knapsack is obtained. By transforming a BKP to the corresponding KP, we actually get many proportionally weighted items, meaning that the problems are very hard to solve.

- Martello and Toth (1990a) do not recognize this problem, since their test instances have capacity $c = \frac{1}{2} \sum_{j=1}^{n} m_j w_j$. For the so-called weakly correlated data instances,

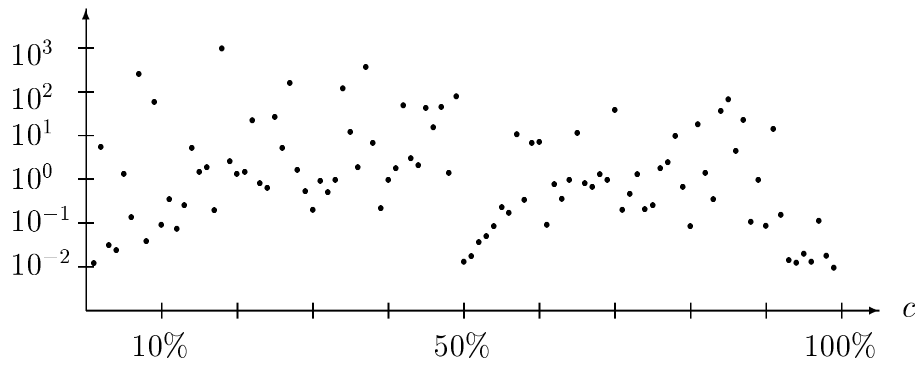

Figure 1: Average computational times for MTB2 in seconds (logarithmic scale), as function of the capacity c. Weakly correlated instances, $n = 2000, R = 100$.

this choice of capacity leads to the easiest possible data instances. Figure 1 shows the actual running times of MTB2 when the capacity is varied from 1% to 99% of the total weight sum. Each point shows the average times of 200 randomly generated data instances with the given capacity. The weights are randomly distributed in $[1, 100]$, while the bounds are distributed in $[1, 10]$. For $c = 50\%$ the running times are measured to 0.01 seconds, while for most other capacities they are hundreds of times larger.

However Figure 1 suffers from the problem, that the exponentially growing computational times of MTB2 make it difficult to ascribe a merit to the mean values. Thus Figure 2 shows the average values of the logarithm to the computational times. Here it is even more clear, that a capacity chosen as 50% of the total weight sum leads to the easiest data instances.

- Better reductions can be achieved by applying specialized reductions for the bounded knapsack problem. If the BKP is transformed to the corresponding KP, each item is reduced separately, meaning that you cannot give a total bound on the number of items of a given type, that may be included in the knapsack.

These observations motivate construction of a specialized algorithm for the BKP, which adapts the latest results from the solution of 0-1 Knapsack Problems and Multiple-Choice Knapsack Problems. The presented algorithm — called BOUKNAP — is based on dynamic

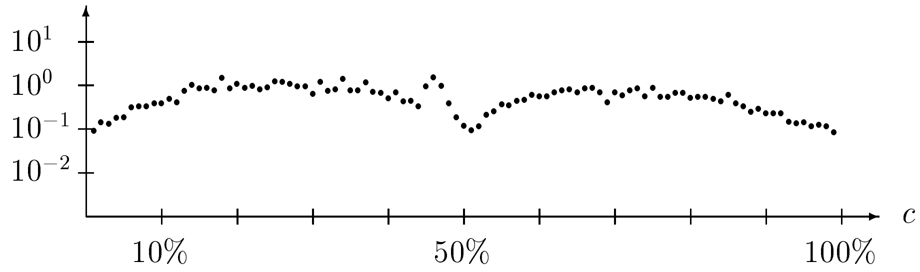

Figure 2: Average log computational times for MTB2 in seconds, as function of the capacity c. Weakly correlated instances, $n = 2000, R = 100$.

programming, where the enumeration starts from the LP-optimal solution and propagates guided by some greedy principles. Since the necessary sorting and reduction is done by need, we are able to establish some minimal properties for the algorithm.

Section 2 gives a brief introduction to minimal algorithms for knapsack problems, while Section 3 brings some basic definitions and sketches the main part of BOUKNAP. A recursive formulation of the dynamic programming is given in Section 4, while Sections 5 and 6 bring some reduction rules for reducing item types, and for fathoming states in the dynamic programming. Finally Section 7 shows how the solution vector corresponding to the objective value is determined. Computational experiments, comparing BOUKNAP with the algorithm MTB2 of Martello and Toth (1990a), are presented in Section 8.

2 Minimal algorithms

Balas and Zemel (1980) presented an efficient algorithm for the solution of the *0-1 Knapsack Problem* by focusing the enumeration on a so-called *core*: A small subset of the items where there is a high probability for finding an optimal solution through permutations. Assume that the items are ordered according to nonincreasing efficiencies $e_j = p_j/w_j$, and let the *break item* b be defined as:

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

Then the core C is basically an interval $[s,t] \ni b$ of the sorted items, such that items $j \notin [s,t]$ may be fixed at their LP-optimal values through reduction tests. The smallest size of a core can obviously not be identified before the problem is solved, since it demands knowledge on the optimal solution, but several *estimates* on the expected core size have been proposed (Balas and Zemel 1980, Martello and Toth 1988,1990a).

However Pisinger (1994c) presented an adaptive technique such that a minimal core can be derived during the solution process. The principles may easily be generalized to the Multiple-Choice Knapsack Problem (Pisinger 1994d), or – as presented here – the Bounded Knapsack Problem.

- The algorithms are based on *dynamic programming* to obtain a pseudopolynomial worst-case behavior.
- The enumeration starts from the *LP-optimal* solution and continues in a symmetric way outwards from the fractional variable of the LP-solution. Thus we enumerate a so-called *expanding core* C.
- In each step of the dynamic programming we branch on the variable which gives the largest gain per weight unit, or the smallest loss per weight unit. This is known as the *greedy approach*.
- Both *sorting* and *reduction* is done by need. When the enumeration reaches the border of the current core, new items are simply reduced, sorted, and added to the core.

- The process stops, when all states of the dynamic programming have been fathomed due to a bounding test, or when all items outside the core could be fixed at their LP-optimal value through reduction rules.

- We would like to minimize all steps of the algorithm: sorting, reduction and enumeration, but due to the computational complexity of each step, an obvious *hierarchy* evolves:

  1 Highest priority is to obtain the the smallest possible core, since enumerating the core has exponential growth $O(2^n)$.

  2 Limiting the so-called strong reduction has second priority, since the complexity of the reduction is $O(r)$ where r is the number of states at the current stage of the dynamic programming.

  3 The smallest possible sorting has third priority, since sorting n items has the complexity $O(n \log n)$.

  4 Lowest priority is to reduce items with the so-called weak upper bound test, since testing n items may be done in $O(n)$.

Since all possible variations of the solution vector in the core have been tested before the core is extended with a new item type, and since the choice of the considered items is guided by greedy principles, the final core C is minimal in the following sense:

Assume that the core was given in advance as an interval $[s, t]$ of the sorted items, like in MT2 by Martello and Toth (1988). In order to solve the problem to proved optimality, we demand that either the enumeration fades out due to some bounding rules before reaching the item types $s-1$ or $t+1$, or that all remaining variables $x_j \notin C$ may be fixed at their LP-optimal values by reduction rules.

But the presented algorithm finds the minimal core, such that an algorithm like MT2 is able to prove the optimality when given the core in advance. This means, that you cannot find a smaller subset core $C' \subset C$ such that a fixed-core algorithm solves the problem to optimality (but you may find a smaller core C' where $C' \not\subset C$ and $C \not\subset C'$ if C' is not symmetrical around b).

The reductions of a minimal algorithm are also the strongest possible, since the bounding rules are not applied before all item types of higher (resp. lower) efficiency have been enumerated, thus ensuring the tightest possible lower bound.

The complexity of a minimal algorithm for the *Bounded Knapsack Problem* is

$$O(n + \min(m_s \cdots m_t,\ nc \log c)), \tag{4}$$

meaning that for moderate core sizes $|C| = t-s+1$, we have a linear-time behavior, while hard instances (demanding a complete enumeration) have a pseudopolynomial solution time.

3 Definitions and main algorithm

The Linear Bounded Knapsack Problem is easily solved by a greedy algorithm: Order the item types according to nonincreasing efficiencies such that

$$e_i \geq e_j \quad \text{if} \quad i < j, \tag{5}$$

and define the *break item type* b as

$$b = \min\left\{ j : \textstyle\sum_{i=1}^{j} m_j w_j > c \right\}. \tag{6}$$

Then an optimal solution to LBKP is defined as $x_j = m_j$ for $j = 1, \ldots, b-1$, and $x_j = 0$ for $j = b+1, \ldots, n$, while we set

$$x_b = \frac{(c - \sum_{j=1}^{b-1} m_j w_j) p_b}{w_b}. \tag{7}$$

The break item type may be found in $O(n)$ time by adapting the technique of Balas and Zemel (1980) or through partial sorting as presented in Pisinger (1994a). Both techniques lead to an $O(n)$ solution time for the LBKP. By truncating the LP-optimal solution to $x_b = 0$ we obtain the *break solution* x', which has profit sum $\overline{p} = \sum_{j=1}^{b-1} m_j p_j$ and weight sum $\overline{w} = \sum_{j=1}^{b-1} m_j w_j$.

The break solution is a tight lower bound, since generally it is only necessary to modify a few variables around b in order to obtain an optimal solution. Figure 3 shows the frequency of item types where the IP-optimal solution differ from the break solution. The graph is a result of solving 1000 randomly generated data instances, constructed such that $b = 500$ for all problems. It is clearly seen that generally only a dozen item types need to be enumerated, and that the probability for changes decreases steeply with a variables distance from b.

This motivates the enumeration of an expanding core C starting from the break item type. The greedy approach for choosing the next variable to be added to the core ensures

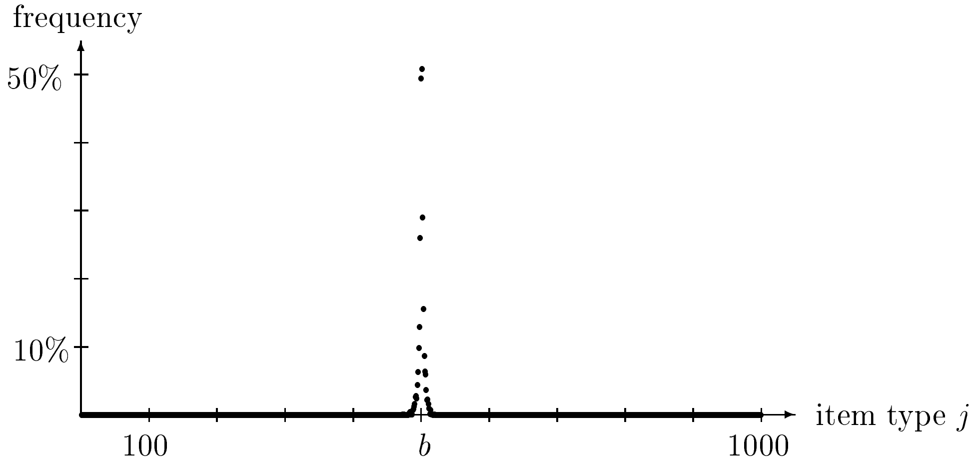

Figure 3: Frequency of item types j where the optimal solution differ from the break solution. Average of 1000 uncorrelated data instances $n = 1000, R = 10000$.

that those variables, where there is largest probability for changes from the LP-optimal values, are enumerated first. For a core $C = [s, t]$ the complete set $X_{s,t}$ of solution vectors is given by

$$X_{s,t} = \left\{ \ (x_s, \ldots, x_t) \ : \ x_j \in \{0, \ldots, m_j\}, \ j = s, \ldots, t \ \right\}, \tag{8}$$

but we will use some fathoming and dominance rules to avoid a complete enumeration.

Sorting and reduction is done by need during the enumeration, thus at any stage $[s', t']$ denotes the interval of sorted items, which have passed the reduction test, while z denotes the current *lower bound*. We may sketch the main part of algorithm BOUKNAP as:

Algorithm 1
procedure bouknap$(n, c, p_1 \ldots p_n, w_1 \ldots w_n, m_1 \ldots m_n, x_1 \ldots x_n)$;
partsort$(1, n)$; {*Find break item type b through partial sorting*}
$[s, t] := [b, b]$; $[s', t'] := [b, b]$;
$z := 0$; $X_{s,t} := \{(0), \ldots, (m_b)\}$;
reduceset$(X_{s,t})$;
while $(X_{s,t} \neq \emptyset)$ **do**
  if $(s - 1 \geq s')$ **then**
    $d :=$ tighten$(s - 1)$;
    if $(d \neq 0)$ **then** $X_{s-1,t} :=$ add$(X_{s,t}, s - 1, d)$; **fi**;
    $s := s - 1$;
  fi;
  reduceset$(X_{s,t})$;
  if $(t + 1 \leq t')$ **then**
    $d :=$ tighten$(t + 1)$;
    if $(d \neq 0)$ **then** $X_{s,t+1} :=$ add$(X_{s,t}, t + 1, d)$; **fi**;
    $t := t + 1$;
  fi;
  reduceset$(X_{s,t})$;
elihw;

The first step of the algorithm is to find the break item type b through partial sorting in $O(n)$ time, as described in Pisinger(1994a): We use a partitioning algorithm like QUICKSORT (Hoare 1962) to partition the variables into two sets around the median λ such that $e_j \geq \lambda$ for $j \in [1, i]$, while $e_j \leq \lambda$ for $j \in [i + 1, n]$. If $\sum_{j=1}^{i} m_j w_j \leq c$, we may conclude that b cannot be in the interval $[1, i]$, and may thus discard the interval. Otherwise we discard the interval $[i+1, n]$. The process is repeated till the current interval only contains b. All discarded intervals are pushed to a stack, since they represent a partial ordering of the items, which will be applied later in the algorithm.

Now the algorithm solves an expanding core $C = [s, t]$, such that initially the core only consists of the break item, and then successively it is extended to the left or to the right. We use dynamic programming for enumerating the core, meaning that procedure ADD enumerates all possible variations of the current core plus item type $s - 1$ or $t + 1$.

Dominance relations and bounding rules ensures that only a limited number of the $m_s \cdots m_t$ possible variations of the core are enumerated. Procedure REDUCESET uses some bounding rules to fathom states, which cannot lead to an improved solution. These rules are further described in Section 5. Furthermore, the procedure updates the lower bound z, and extends the core by need.

Procedure TIGHTEN(j) tightens the bound $x_j \in \{0, \ldots, m_j\}$ by applying some bounding rules on the number of items inserted/removed of the concerned item type. Thus the obtained bound d means that only $d \leq m_j$ item values have to be enumerated in procedure ADD, and if $d = 0$ we may even fathom the variable j. The procedure TIGHTEN is further described in Section 6.

Algorithm BOUKNAP terminates when all states $X_{s,t}$ have been fathomed, meaning that a better solution than the current cannot be found.

4 Dynamic programming

Let $f_i(c)$ be an optimal solution to the following subproblem of BKP, defined on the first i variables of the problem:

$$f_i(c) = \max \left\{ \sum_{j=1}^{i} p_j x_j : \sum_{j=1}^{i} w_j x_j \leq c,\ x_j \in \{0, \ldots, m_j\} \text{ for } j = 1, \ldots, i \right\}. \tag{9}$$

Generalizing the results by Bellman (1957) for the 0-1 Knapsack Problem, we obtain the following recursion formula for the solution of BKP

$$f_i(c) = \max \begin{cases} f_{i-1}(c) \\ f_{i-1}(c - w_i) + p_i \\ \vdots \\ f_{i-1}(c - m_i w_i) + m_i p_i \end{cases} \tag{10}$$

while we set

$$f_0(c) = 0 \text{ for all } c \geq 0. \tag{11}$$

By changing the recursion to an iterative process, Gilmore and Gomory (1966) and Nemhauser and Ullmann (1969) developed dynamic programming algorithms with pseudopolynomial time bounds. However, none of these algorithms are able to solve large instances: Nemhauser and Ullmann (1969) report that the largest instances solved were $n = 50$ with $m_j = 2$ for all item types.

The problem about these algorithms is that the enumeration has no strategy, since the recursion formula may be evaluated without the sorting (5), resulting in an arbitrary enumeration. However, if we order the item types in advance, the enumeration gets even worse: We start the enumeration at those item types $(1, 2, \ldots)$ where there is least probability (cf. figure 3) for changing the solution variables from their LP-optimal values — generating billions of states before the problem even starts to be interesting.

A more efficient recursion should take into account, that generally only a few items around b need to be changed from their LP-optimal values in order to obtain the IP-optimal values. Assume that the items are ordered according to nonincreasing efficiencies, and let $f_{s,t}(c)$ $(s \le b \le t)$ be an optimal solution to the core problem:

$$f_{s,t}(c) = \max \left\{ \begin{array}{l} \sum_{j=1}^{s-1} m_j p_j + \sum_{j=s}^{t} p_j x_j : \\ \sum_{j=1}^{s-1} m_j w_j + \sum_{j=s}^{t} w_j x_j \le c, \\ x_j \in \{0, \ldots, m_j\} \text{ for } j = s, \ldots, t \end{array} \right\}. \tag{12}$$

This leads to the following improved recursion:

$$f_{s,t}(c) = \max \left\{ \begin{array}{ll} f_{s,t-1}(c) & \text{if } t \ge b \\ f_{s,t-1}(c - w_t) + p_t & \text{if } t \ge b \\ \vdots & \\ f_{s,t-1}(c - m_t w_t) + m_t p_t & \text{if } t \ge b \\ f_{s+1,t}(c) & \text{if } s < b \\ f_{s+1,t}(c + w_s) - p_s & \text{if } s < b \\ \vdots & \\ f_{s+1,t}(c + m_s w_s) - m_s p_s & \text{if } s < b \end{array} \right. \tag{13}$$

while we set

$$f_{b,b-1}(c) = \sum_{j=1}^{b-1} m_j p_j \text{ for all } c \ge \sum_{j=1}^{b-1} m_j w_j. \tag{14}$$

Thus the enumeration starts at $(s,t) = (b, b-1)$ and continues by either removing some items of type s from the knapsack, or inserting some items of type t in the knapsack. An optimal solution to BKP is found as $f_{1,n}(c)$.

At any stage of the enumeration, it is convenient to represent the states by the corresponding profit and weight sums (π, μ), thus with a core $C = [s,t]$ the set of states is $X_{s,t} = \{(\pi_1, \mu_1), \ldots, (\pi_r, \mu_r)\}$, where

$$\pi_i = f_{s,t}(\mu_i). \tag{15}$$

It is convenient to keep the states ordered, such that $X_{s,t} = \{(\pi_1, \mu_1), \ldots, (\pi_r, \mu_r)\}$, satisfies $\pi_i < \pi_{i+1}$ and $\mu_i < \mu_{i+1}$. Any state (π, μ) with $\mu > c + \sum_{j=1}^{s-1} m_j w_j$, may be fathomed, since even if we removed all items of types $j < s$ in the forthcoming iterations, the state would never become a feasible solution. This observation implies that no states with weights $\mu \ge 2c$ can occur, giving the algorithm space complexity $O(n2c) = O(nc)$.

An efficient iterative version of recursion (13) is obtained by applying the transformation (2) for the current item type, such that the procedure ADD in algorithm 1 results in $\lfloor \log_2 m_j + 1 \rfloor$ mergings of length $O(c)$ (see Pisinger (1994c) for details on the merging). This means that the time complexity for the dynamic programming is $O(c \sum_{j=1}^{n} \lfloor \log_2 m_j + 1 \rfloor)$, i.e. $O(nc \log c)$ in the worst case.

5 Reduction

Although the recursion (13) gives a pseudopolynomial time bound on the solution process, even less enumeration may be done by incorporating some bounding rules in the dynamic programming. Assume that the current core is $C = [s, t]$, and that a given state i has the profit and weight sum (π_i, μ_i). We may then fathom the state if the following condition holds:

$$u(i) < z + 1, \tag{16}$$

where the upper bound $u(i)$ on the stage is obtained by relaxing the constraints on x_{s-1} and x_{t+1} to $x_{s-1} \geq 0$ and $x_{t+1} \geq 0$, yielding:

$$u(i) \quad = \quad \begin{cases} \pi_i + \dfrac{(c - \mu_i) \cdot p_{t+1}}{w_{t+1}} & \text{if} \quad \mu_i \leq c, \\ \\ \pi_i + \dfrac{(c - \mu_i) \cdot p_{s-1}}{w_{s-1}} & \text{if} \quad \mu_i > c. \end{cases} \tag{17}$$

This bound may also be used for deriving a global upper bound on (1) as

$$u_{\text{BKP}} = \max_{i \in X_{s,t}} u(i), \tag{18}$$

which has the property that it corresponds to the optimal solution when $(s, t) = (1, n)$.

Since $s - 1$ or $t + 1$ in (17) may fall outside the current reduced and sorted interval $[s', t']$, we have to extend the core in such cases. This is done the following way:

- If all items $j < s'$ have been fathomed due to (16) we choose $p_{s-1} = \infty$ and $w_{s-1} = 1$. Similarly if all items $j > t'$ have been fathomed we choose $p_{t+1} = 0$ and $w_{t+1} = 1$. In both cases the bounds will ensure that states which cannot be improved further are fathomed.

- Otherwise we choose one of the partially ordered intervals discarded by procedure PARTSORT and try to fix as many variables x_j as possible, at their optimal values.

An item type j may be fathomed if either all the available items of that type have to be included in the knapsack ($j < b$) or if none of the available items of that type can be included in the knapsack ($j \geq b$). Thus we may use the following fathoming tests

$$\begin{aligned} \overline{p} + p_j + \frac{(c - \overline{w} - w_j)p_b}{w_b} &< z + 1 \quad \text{if} \quad j < b, \\ \overline{p} - p_j + \frac{(c - \overline{w} + w_j)p_b}{w_b} &< z + 1 \quad \text{if} \quad j \geq b. \end{aligned} \tag{19}$$

If the inequality is satisfied, we may fix x_j at m_j ($j < b$), respectively at 0 ($j \geq b$), and thus fathom the item type. The remaining variables are swapped to positions besides $[s', t']$, where they are sorted, and added to the core. See Pisinger (1994c) for more details.

6 Tightening the bound on an item type

Since the enumeration of a new item type is computationally very expensive, a strong bounding test should be used for tightening the bound on an item type ℓ. In this way the bound may be restricted from $x_\ell \in \{0, \ldots, m_\ell\}$ to $x_\ell \in \{0, \ldots, d_\ell\}$ if $\ell \geq b$ or to $x_\ell \in \{m_\ell - d_\ell, \ldots, m_\ell\}$ if $\ell < b$. Thus assume that the core is $C = [s, t]$ and that we are going to add item type ℓ to the core ($\ell = s - 1$ or $\ell = t + 1$).

It is only fruitful to insert d items of type $\ell = t+1$ in the knapsack if any upper bound u on the BKP with additional constraint $x_\ell = d$ exceeds the current lower bound z, thus if

$$u_\ell(d) \geq z + 1. \tag{20}$$

We use a generalization of the bound by Dembo and Hammer (1980) for this test, obtaining

$$\overline{p} + dp_\ell + \frac{(c - \overline{w} - dw_\ell)p_b}{w_b} \geq z + 1. \tag{21}$$

From this inequality we may obtain the maximum number of an item type, which may be included to the knapsack, as:

$$d_\ell = \left\lfloor \frac{\det(z + 1 - \overline{p}, c - \overline{w}, p_b, w_b)}{\det(p_\ell, w_\ell, p_b, w_b)} \right\rfloor \quad \text{when} \quad \ell = t + 1, \tag{22}$$

where we set $d_\ell = m_\ell$ when $e_\ell = e_b$ or when the right side of equation (22) is larger than m_ℓ. A similar result is obtained for items of type $\ell = s - 1$, which have to be removed from the knapsack. Here an upper bound on the number of removed items is

$$d_\ell = \left\lfloor \frac{\det(z + 1 - \overline{p}, c - \overline{w}, p_b, w_b)}{-\det(p_\ell, w_\ell, p_b, w_b)} \right\rfloor \quad \text{when} \quad \ell = s - 1, \tag{23}$$

with the same conventions as for equation (22).

Although these bounds works well for most instances, tighter reductions may be derived by using enumerative bounds as presented in Pisinger (1994c). Thus with the core $C = [s, t]$ let $X = X_{s,t}$ be the set of enumerated solution vectors. The ultimate check is to determine whether any state in $X + \ell$ will pass the reduction (17) when d items of type ℓ are added to the knapsack. No stronger bound can be constructed, since if the inclusion of the d items implies that a new promising branch is introduced to $X_{s,t}$ there is no way of avoiding the inclusion.

If we add d items of item type $\ell = t + 1$, any state in $X + \ell$ will be the sum $(\pi_i + dp_\ell, \mu_i + dw_\ell)$ where $0 \leq d \leq m_\ell$, meaning that the bound (17) for fathoming states in $X + \ell$ becomes:

$$\tilde{u}(i, d) = \begin{cases} \tilde{u}_1(i, d) = \pi_i + dp_\ell + \dfrac{(c - \mu_i - dw_\ell)p_{t+1}}{w_{t+1}} & \text{if } \mu_i + dw_\ell \leq c, \\[2ex] \tilde{u}_2(i, d) = \pi_i + dp_\ell + \dfrac{(c - \mu_i - dw_\ell)p_{s-1}}{w_{s-1}} & \text{if } \mu_i + dw_\ell > c. \end{cases} \tag{24}$$

Thus an upper bound for the addition of d items of type ℓ is given as

$$\tilde{u}_{\ell,d} = \max_{i \in X} \tilde{u}(i,d), \tag{25}$$

where we demand that $\tilde{u}_{\ell,d} \geq z+1$. This is a generalization of the π-bound presented in Pisinger (1994b). From equation (24) we may derive the maximum amount d of items of the type ℓ, which may be added to the knapsack such that state i passes the criteria $\tilde{u}(i,d) \geq z+1$. The equation

$$\tilde{u}_1(i,d) \geq z+1 \quad \text{if} \quad \mu_i + dw_\ell \leq c, \tag{26}$$

may be written

$$\det(\pi_i - z - 1, \mu_i - c, p_{t+1}, w_{t+1}) \geq d \det(p_{t+1}, w_{t+1}, p_\ell, w_\ell) \quad \text{if} \quad d \leq \frac{c-\mu_i}{w_\ell}. \tag{27}$$

Here the left side is positive, since any state $i \in X$ has passed the reduction (16), and the right side is zero, since $\ell = t+1$. So the equation is satisfied for all states i when just $d \leq \frac{c-\mu_i}{w_\ell}$. The largest contribution we may get from this restriction is thus

$$\max_{i \in X} \left(\frac{c-\mu_i}{w_\ell} \right) = \frac{c-\mu_1}{w_\ell}, \tag{28}$$

since the states in X are ordered, meaning that μ_1 is the smallest weight sum. In a similar way the equation

$$\tilde{u}_2(i,d) \geq z+1 \quad \text{if} \quad \mu_i + dw_\ell > c, \tag{29}$$

may be formulated as

$$d \leq \frac{\det(\pi_i - z - 1, \mu_i - c, p_{s-1}, w_{s-1})}{-\det(p_\ell, w_\ell, p_{s-1}, w_{s-1})} \quad \text{if} \quad d > \frac{c-\mu_i}{w_\ell}. \tag{30}$$

Those states, which does not satisfy that $d > \frac{c-\mu_i}{w_\ell}$, will not contribute to the bound d_ℓ since they are dominated by the bound (28), while those states satisfying the inequality will contribute with

$$d'(i) = \frac{\det(\pi_i - z - 1, \mu_i - c, p_{s-1}, w_{s-1})}{-\det(p_\ell, w_\ell, p_{s-1}, w_{s-1})}. \tag{31}$$

Thus the tightened bound on item type ℓ is

$$d_\ell = \max\left\{ \lfloor \frac{c-\mu_1}{w_\ell} \rfloor, \; \lfloor \max_{i \in X} d'(i) \rfloor \right\}, \quad \text{when} \quad \ell = t+1, \tag{32}$$

where we set $d_\ell = m_\ell$ if the right side of equation (32) is larger than m_ℓ.

A similar result is obtained for $\ell = s-1$, resulting in the tightened bound

$$d_\ell = \max\left\{ \lfloor \frac{\mu_r - c}{w_\ell} \rfloor, \; \lfloor \max_{i \in X} d''(i) \rfloor \right\}, \quad \text{when} \quad \ell = s-1. \tag{33}$$

Here μ_r is the largest weight sum in X, while $d''(i)$ is given by

$$d''(i) = \frac{\det(\pi_i - z - 1, \mu_i - c, p_{t+1}, w_{t+1})}{\det(p_\ell, w_\ell, p_{t+1}, w_{t+1})}. \tag{34}$$

Note, that if $\frac{c-\mu_1}{w_\ell} \geq m_\ell$ in (32) we do not need to evaluate the enumerative bound, since there is no hope for tightening the bound m_ℓ. A similar rule holds for (33) when $\frac{\mu_m - c}{w_\ell} \geq m_\ell$. Both rules may save a considerable amount of computational time.

Computational experiments have shown, that the bounds (32) and (33) are able to tighten the bound on item type ℓ by 10% more than the bounds (22) and (23). For practical purpose, the simpler bounds may be more useful, especially since they are evaluated in constant time. This paper is however focused on determining the smallest possible core in order to get a tight bound on the enumeration.

7 Solution vector

Any state (π_i, μ_i) in the dynamic programming should not only contain the profit and weight sum, but also the corresponding solution vector $\mathbf{x}$. Pisinger (1994c) noticed that since only a few item types around b are modified during the solution process, storing a complete solution vector $\mathbf{x}$ would be too comprehensive. Instead only the last a changes in the solution vector are saved. When an optimal solution is found, we reconstruct the solution vector. If more than a changes were made, we have to solve a new (smaller) problem to find the remaining vector. The process is eventually repeated. In this way, we get a space bound $O(c)$ for the algorithm. For a complete description of the technique see Pisinger (1994c).

8 Computational experiments

The presented algorithm has been implemented in ANSI-C, and a complete listing is available from the author on request. The following results have been achieved on a HP9000/730 computer.

We will consider the most common randomly generated data instances from the literature: *Uncorrelated data instances* (UC): p_j and w_j are randomly distributed in $[1, R]$. *Weakly correlated data instances* (WC): w_j randomly distributed in $[1, R]$ and p_j randomly distributed in $[w_j - R/10, w_j + R/10]$ such that $p_j \geq 1$. *Strongly correlated data instances* (SC): w_j randomly distributed in $[1, R]$ and $p_j = w_j + 10$. *Subset-sum data instances* (SS): w_j randomly distributed in $[1, R]$ and $p_j = w_j$. The *data range* R will be tested with values $R = 100, 1000$ and 10000.

These data instances are generated as described in Pisinger (1994e) appendix B, with the addition that the bounds m_j are randomly distributed in $[5, 10]$ (see Appendix A). Each problem type is tested with a series of $S = 200$ instances, such that the capacity c of the knapsack varies from 1% to 99% of the total weight sum of the items. This approach

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	4	5	6	5	9	9	32	36	33	4	5	6
300	5	7	7	4	10	13	97	99	102	4	5	6
1000	4	8	10	4	9	16	287	296	311	4	5	6
3000	4	9	12	6	7	18	991	956	943	4	5	6
10000	4	6	15	11	5	17	3142	3404	—	4	5	6
30000	7	5	15	22	6	10	9282	9164	—	4	5	6
100000	18	5	11	60	10	7	29621	30080	—	4	5	6

Table I: Final core size, measured as $|C| = t - s + 1$. Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	65	88	91	40	95	97	86	90	91	13	16	20
300	40	84	88	14	86	98	87	86	87	6	7	8
1000	6	71	87	4	49	97	82	84	88	2	3	3
3000	2	51	84	2	13	92	87	86	86	1	1	2
10000	1	15	76	1	2	69	84	87	—	0	0	0
30000	0	3	59	1	0	24	84	85	—	0	0	0
100000	0	0	28	0	0	3	84	85	—	0	0	0

Table II: Percentage of items tested by the weak bound. Average of 200 instances.

takes into account that the running times depend on the chosen capacity as shown in Figure 1.

We will compare the presented BOUKNAP algorithm with the MTB2 algorithm by Martello and Toth (1990a), which in the same paper is shown to be the most efficient of all published algorithm. The FORTRAN code for MTB2 has been obtained from Martello and Toth (1990a). Since MTB2 transforms the BKP to an equivalent 0-1 KP, instances larger than $n = 50.000$ cannot be solved due to memory limitations. In the following tables a "—" means, that the 200 instances could not be solved in 10 hours.

First, Table I shows the average core size for each problem type, as obtained by BOUKNAP. It is seen that only a couple of item types need to be enumerated in order to solve the problem to optimality. However the strongly correlated instances demand a considerable enumeration. The core sizes are minimal in the sense, that for each instance one cannot find any *subset* of the core, such that the problem is solved to proved optimality

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	86	90	90	71	84	83	47	45	48	18	14	11
300	91	95	95	68	93	92	50	48	48	18	25	9
1000	85	98	98	57	95	97	52	53	53	16	22	11
3000	81	99	99	58	94	99	50	50	52	19	8	11
10000	71	99	100	17	91	100	52	48	—	23	26	9
30000	36	97	100	10	76	100	53	54	—	15	10	17
100000	71	85	100	15	74	99	54	54	—	14	21	8

Table III: Efficiency of weak reduction in percent. Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	67	71	70	57	68	69	65	64	64	13	10	7
300	63	69	71	38	68	72	64	64	64	14	11	8
1000	37	70	71	21	66	70	64	65	66	12	11	6
3000	26	69	71	13	52	73	64	65	64	12	12	7
10000	16	56	75	6	20	73	64	65	—	12	12	7
30000	9	30	72	3	10	61	65	65	—	13	11	7
100000	3	12	77	1	5	28	65	64	—	11	12	8

Table IV: Efficiency of strong reduction in percent. Average of 200 instances.

by considering only the items in the core.

Table II gives the percentage of items, which are tested by the weak bound (19). We observe that for large data instances, only a minor part of the item types need to be considered at all, proving the strength of the lazy reduction.

The next two tables show the efficiency of the presented reductions: Table III shows the efficiency of reduction test (19), by stating the amount of tested item types, which were reduced (in percent). The efficiency of the tightening of bounds (33) is given in Table IV, measured as $(\sum_j(m_j - d_j))/(\sum_j m_j)$.

Finally Table V shows the maximum amount of states at any stage of the dynamic programming. The number of states increases with larger data-range R and problem size n. However due to the compact representation of the solution vector $\mathbf{x}$, even large strongly correlated instances may be solved.

The last two tables (Table VI and VII) compares the running times of BOUKNAP with those of MTB2. Notice, that MTB2 has substantial stability problems for low-ranged data instances even of relatively small size ($n = 3000$). Also subset-sum data instances shows a few anomalous occurrences. On the other hand BOUKNAP has a very stable behavior, solving most instances within a fraction of a second. The strongly correlated instances demand more computational time, but the effort grows linearly with the problem size n and the data range R. Similar tests have been performed with data instances generated such that m_j is randomly generated in $[50, 100]$. The computational times for these instances show the same behavior as in Table VI, but are about 5 times larger.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	1	1	1	1	3	5	5	44	560	3	24	279
300	1	2	2	2	9	15	6	61	707	2	27	294
1000	1	7	7	3	16	21	11	116	1022	3	28	290
3000	2	10	20	6	30	55	22	191	1788	4	39	261
10000	3	12	28	9	33	114	37	361	—	5	25	255
30000	4	17	83	26	46	199	80	555	—	4	29	257
100000	23	25	127	61	65	289	242	999	—	3	23	276

Table V: Max number of states in the dynamic programming (in thousands). Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.16	1.55	0.00	0.02	0.27
300	0.00	0.00	0.00	0.00	0.01	0.01	0.06	0.62	7.36	0.00	0.02	0.27
1000	0.00	0.01	0.01	0.00	0.01	0.03	0.22	2.24	25.70	0.00	0.02	0.28
3000	0.00	0.01	0.02	0.01	0.01	0.09	0.81	9.37	109.67	0.01	0.03	0.30
10000	0.01	0.02	0.06	0.02	0.03	0.16	3.03	39.71	—	0.02	0.04	0.30
30000	0.04	0.05	0.12	0.08	0.08	0.20	13.59	116.96	—	0.05	0.06	0.31
100000	0.20	0.16	0.27	0.40	0.26	0.37	107.90	450.46	—	0.17	0.19	0.45

Table VI: Total computing time in seconds (BOUKNAP). Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	0.00	0.00	0.00	0.00	0.01	0.02	—	—	—	0.00	0.02	0.15
300	0.01	0.01	0.01	0.00	0.03	0.06	—	—	—	0.00	—	40.96
1000	0.01	0.03	0.04	0.01	0.05	0.26	—	—	—	0.00	0.01	0.09
3000	0.02	0.05	0.14	62.58	0.04	0.57	—	—	—	0.01	0.02	0.10
10000	14.72	0.11	0.43	—	0.17	0.98	—	—	—	0.05	0.06	0.14
30000	—	0.31	1.09	—	—	1.04	—	—	—	0.18	0.19	0.26
100000	—	—	—	—	—	—	—	—	—	—	—	—

Table VII: Total computing time in seconds (MTB2). Average of 200 instances.

It is interesting to compare Table VII with Table II, since MTB2 has the worst computational times in those situations, where fewest items need to be considered in order to solve the problem. This demonstrates that MTB2 does not fully utilize the benefits of solving a core problem, since MTB2 spends more time for solving the core problem than a complete sorting and reduction would take.

9 Conclusions

We have presented a complete algorithm for the exact solution of the Bounded Knapsack Problem, which has the time bound

$$O(n + \min(m_s \cdots m_t, nc \log c)), \tag{35}$$

meaning that the solution time is linear for small core sizes $|C| = t - s + 1$. This result is strengthened by the fact that the presented algorithm enumerates the smallest possible core. Since the reduction and sorting is done by need, these preprocessing steps are also minimized, although they have lower priority than the enumeration.

Moreover we have demonstrated that the transformation approach which was proposed by Martello and Toth (1990a) leads to hard data instances, even for relatively small instances ($n = 3000$). The presented algorithm does not have these problems, and computational experiments document, that BOUKNAP outperforms MTB2 for all kinds of randomly generated problem types.

Finally it should be mentioned that BOUKNAP is well suited as an approximate algorithm, since the enumeration may be terminated when the global upper bound u_{BKP} differs less than a given threshold value from the current solution.

Appendix A: Generating test data

In order to let other authors generate the same test instances as applied in Section 8, we here include an algorithm for generating the test data.

The standard LRAND48 generator of the C library is used for generating pseudo-random profits and weights. The LRAND48 generator is using the well-known linear congruential algorithm, which generates a series of numbers $X_1, X_2, X_3, \ldots$ as

$$X_{i+1} = (aX_i + d) \bmod m, \tag{36}$$

where $a = 25214903917$, $d = 11$ and $m = 2^{48}$. The first 31 bits of the number are returned for LRAND48. A seed s for the algorithm is given by procedure SRAND48 as:

$$X_0 = s \cdot 2^{16} + 13070. \tag{37}$$

For a given data range R, bound range M, instance size n, and problem type t (uc, wc, sc, ss) we constructed $S = 200$ different data instances as follows:

```
procedure testinstance(n, M, R, t, i);  {Generate test instance i, 1 ≤ i ≤ S}
W := 0; R' := ⌊R/10⌋;
SRAND48(i);  {Use i as seed for the sequence of random numbers}
for j := 1 to n do
  w_j := (LRAND48 mod R) + 1;
  m_j := (LRAND48 mod M/2) + M/2;
  case t of
    uc: p_j := (LRAND48 mod R) + 1;
    wc: p_j := w_j − R' + (LRAND48 mod (2R' + 1));
          if (p_j ≤ 0) then p_j := 1; fi;
    sc: p_j := w_j + 10;
    ss: p_j := w_j;
  esac;
  W := W + m_j w_j;
rof;
c := ⌊(i * W)/(S + 1)⌋;
if (c ≤ R) then c := R + 1; fi;
for j := 1 to n do
  if (m_j w_j > c) then m_j := ⌊c/w_j⌋; fi;
rof;
```

The **if** -statement in the weakly correlated instances ensures that p_j is a positive integer, while the last lines of the algorithm ensures that $m_j w_j \le c$ and $m_j \ge 1$ for all $j = 1, \ldots, n$. Checksums of the optimal solutions and capacities are given in Table VIII and IX.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	715	95	435	893	695	148	510	871	835	692	311	400
300	263	897	650	823	404	45	856	910	286	436	438	514
1000	793	818	751	698	10	543	663	196	648	783	256	758
3000	692	161	184	699	735	33	369	864	251	959	734	761
10000	931	75	120	325	900	778	809	429	—	269	429	830
30000	347	958	62	337	267	927	132	662	—	922	792	703
100000	515	904	833	748	897	214	407	922	—	407	672	998

Table VIII: Checksums for optimal solutions given as $\sum_{i=1}^{S} z_i \bmod 1000$.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	85	253	455	85	253	455	692	311	400	692	311	400
300	981	138	395	981	138	395	436	438	514	436	438	514
1000	848	95	634	848	95	634	783	256	758	783	256	758
3000	4	908	263	4	908	263	959	734	761	959	734	761
10000	898	295	741	898	295	741	269	429	—	269	429	830
30000	431	15	109	431	15	109	922	792	—	922	792	703
100000	15	477	669	15	477	669	407	672	—	407	672	998

Table IX: Checksums for applied capacities given as $\sum_{i=1}^{S} c_i \bmod 1000$.

Appendix B: Additional computational results

In this section we bring the results of some computational experiments which did not fit into the main section.

First, Table X brings the number of iterations, which are needed to define the complete solution vector as described in Section 7. For easy data instances, less than two iterations are needed on the average, meaning that there is a minimal overhead for this part of the algorithm. For strongly correlated instances, up to a dozen iterations are needed, but still less than 10% of the solution time is spent for reconstructing the solution vector.

Table XI shows the gap Γ between the LP-optimal and the IP-optimal solution. Balas and Zemel (1980) showed that the hardness of a 0-1 Knapsack Problem depends on the

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.0	1.0	1.0	1.0
300	1.0	1.0	1.0	1.0	1.0	1.1	1.3	1.3	1.3	1.0	1.0	1.0
1000	1.0	1.0	1.0	1.0	1.1	1.3	1.9	1.8	1.9	1.0	1.0	1.0
3000	1.0	1.0	1.1	1.1	1.0	1.4	2.6	3.0	2.9	1.0	1.0	1.0
10000	1.0	1.0	1.3	1.5	1.0	1.5	2.8	5.1	—	1.0	1.0	1.0
30000	1.2	1.0	1.2	1.8	1.1	1.2	2.9	7.6	—	1.0	1.0	1.0
100000	1.6	1.0	1.1	2.2	1.5	1.0	2.9	11.6	—	1.0	1.0	1.0

Table X: Number of iterations used for obtaining the solution vector. Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	2.8	35.7	364.2	0.4	10.7	110.0	4.5	5.0	4.7	0.0	0.0	0.0
300	1.1	15.4	153.7	0.1	3.6	51.5	4.5	4.6	4.8	0.0	0.0	0.0
1000	0.1	5.9	63.5	0.0	1.1	18.6	4.2	4.2	4.4	0.0	0.0	0.0
3000	0.0	2.1	28.6	0.0	0.3	7.2	4.6	4.7	4.4	0.0	0.0	0.0
10000	0.0	0.5	9.7	0.0	0.1	2.3	4.3	4.9	—	0.0	0.0	0.0
30000	0.0	0.1	3.7	0.0	0.0	0.5	4.3	4.4	—	0.0	0.0	0.0
100000	0.0	0.0	1.0	0.0	0.0	0.1	4.1	4.2	—	0.0	0.0	0.0

Table XI: Gap Γ between LP-optimal solution and IP-optimal solution. Average of 200 instances.

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.20	2.35	0.00	0.02	0.26
300	0.00	0.00	0.00	0.00	0.01	0.01	0.06	0.74	8.80	0.00	0.02	0.24
1000	0.00	0.01	0.01	0.00	0.01	0.02	0.27	2.93	31.81	0.00	0.02	0.24
3000	0.00	0.01	0.01	0.01	0.01	0.06	0.87	11.21	133.46	0.00	0.03	0.25
10000	0.01	0.01	0.03	0.01	0.02	0.09	3.55	46.41	—	0.01	0.02	0.24
30000	0.02	0.01	0.07	0.09	0.05	0.13	14.46	158.89	—	0.01	0.02	0.25
100000	0.30	0.02	0.10	0.71	0.14	0.22	113.94	615.35	—	0.01	0.02	0.24

Table XII: Standard deviation computational times BOUKNAP. Average of 200 instances.

correlation and the gap Γ. This explains that instances with coefficients generated in a large range R, generally are harder to solve than the same instances where the coefficients are generated in a small range, since the gap Γ grows with increasing data range.

Finally Table XII and XIII shows the standard deviation on the running times of BOUKNAP and MTB2. Apart from the strongly correlated instances — which apparently have a very large variation in the running times — most of the variations are very small. However MTB2 has a very large variation at the anomalous occurrences, showing that it is only a few instances, which takes up almost all the computational time.

Data instances, where the bounds m_j are distributed in $[50, 100]$, are considered in Table XIV and XV. Running times for both of the algorithms are given for the data ranges

$n \setminus R$	UC 100	UC 1000	UC 10000	WC 100	WC 1000	WC 10000	SC 100	SC 1000	SC 10000	SS 100	SS 1000	SS 10000
100	0.00	0.01	0.01	0.00	0.02	0.02	—	—	—	0.00	0.08	0.23
300	0.07	0.01	0.01	0.00	0.03	0.07	—	—	—	0.00	—	576.25
1000	0.01	0.02	0.04	0.04	0.05	0.26	—	—	—	0.00	0.01	0.10
3000	0.01	0.03	0.14	631.67	0.03	0.60	—	—	—	0.00	0.01	0.13
10000	205.07	0.04	0.39	—	0.73	0.91	—	—	—	0.00	0.02	0.12
30000	—	0.07	0.86	—	—	1.40	—	—	—	0.01	0.03	0.13
100000	—	—	—	—	—	—	—	—	—	—	—	—

Table XIII: Standard deviation computational times MTB2. Average of 200 instances.

	UC		WC		SC		SS	
$n \setminus R$	100	1000	100	1000	100	1000	100	1000
100	0.00	0.00	0.00	0.01	0.04	0.49	0.03	0.34
300	0.00	0.01	0.01	0.03	0.15	1.88	0.03	0.33
1000	0.00	0.01	0.01	0.06	0.62	6.42	0.03	0.34
3000	0.01	0.03	0.02	0.09	2.96	20.55	0.04	0.31
10000	0.03	0.07	0.13	0.14	23.09	—	0.05	0.38
30000	0.16	0.12	0.51	0.24	175.71	—	0.07	0.37
100000	1.05	0.29	3.78	0.58	—	—	0.20	0.57

Table XIV: Total computing time in seconds (BOUKNAP) when m_j is distributed in $[50, 100]$. Average of 200 instances.

	UC		WC		SC		SS	
$n \setminus R$	100	1000	100	1000	100	1000	100	1000
100	0.00	0.01	0.15	0.03	—	—	35.18	67.45
300	0.22	0.02	0.46	0.06	—	—	—	—
1000	0.02	0.05	41.36	0.07	—	—	0.01	0.02
3000	0.04	0.09	—	0.33	—	—	0.03	0.03
10000	0.95	0.22	—	0.22	—	—	0.10	0.12
30000	—	0.55	—	5.61	—	—	0.33	0.34
100000	—	—	—	—	—	—	—	—

Table XV: Total computing time in seconds (MTB2) when m_j is distributed in $[50, 100]$. Average of 200 instances.

$R = 100$ and $R = 1000$ (range $R = 10000$ causes overflow in the integer arithmetics). As in the previous case, it is seen that MTB2 has several anomalous occurrences, even for small-ranged instances. On the other hand BOUKNAP has a stable behavior, solving all problems in a few seconds (apart from the strongly correlated instances). It should especially be noted, that solving a BKP which involves up to 100000 items (1000 item types with bounds up to 100) is considerably more efficient than solving a 0-1 Knapsack Problem of similar size (Pisinger 1994c). Thus in cases, where several items of same profit and weight occur, a transformation from the 0-1 Knapsack Problem to a Bounded Knapsack Problem will yield a considerable decrease in the computational times.

References

BALAS, E. AND E. ZEMEL, "An Algorithm for Large Zero-One Knapsack Problems," *Operations Research*, 28 (1980), 1130-1154.

BULFIN R.L, R.G. PARKER AND C.M. SHETTY, "Computational results with a branch and bound algorithm for the general knapsack problem," *Naval Research Logistics Quarterly*, 26 (1979), 41-46.

BELLMAN, R.E., "Dynamic Programming," *Princeton University Press, Princeton, N.J.*, (1957).

Dembo, R.S. and P.L. Hammer, "A Reduction Algorithm for Knapsack Problems," *Methods of Operations Research*, 36 (1980), 49-60.

Gilmore, P.C. and R.E. Gomory, "The theory and computation of knapsack functions," *Operations Research*, 14 (1966), 1045-1074.

Hoare, C.A.R., "Quicksort," *Computer Journal*, 5 (1962), 10-15.

Ingargiola, G.P. and J.F. Korsh, "A general algorithm for the one-dimensional knapsack problem," *Operations Research*, 25 (1977), 752-759.

Martello, S. and P. Toth, "Branch and bound algorithms for the solution of general unidimensional knapsack problems," In M.Roubens (ed.), *Advances in Operations Research*, North-Holland, Amsterdam (1977), 295-301.

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

Martello, S. and P. Toth, Knapsack Problems: Algorithms and Computer Implementations, Wiley, Chichester, England, (1990a).

Martello, S. and P. Toth, "An exact algorithm for large unbounded knapsack problems," *Operations Research Letters*, 9 (1990b), 15-20.

Nemhauser, G.L. and Z. Ullmann, "Discrete dynamic programming and capital allocation," *Management Science*, 15 (1969), 494-505.

Pisinger, D., "An expanding-core algorithm for the exact 0-1 Knapsack Problem," *European Journal of Operational Research* (1994a), To appear.

Pisinger, D., "Solving hard knapsack problems," *DIKU, University of Copenhagen, Denmark,* Report 94/24 (1994b).

Pisinger, D., "A minimal algorithm for the 0-1 Knapsack Problem," *DIKU, University of Copenhagen, Denmark,* Report 94/23 (1994c).

Pisinger, D., "A minimal algorithm for the Multiple-Choice Knapsack Problem," *DIKU, University of Copenhagen, Denmark,* Report 94/25 (1994d).

Pisinger, D., "Core problems in Knapsack Algorithms," *DIKU, University of Copenhagen, Denmark,* Report 94/26 (1994e).