A minimal algorithm for the 0-1 Knapsack Problem.*

David Pisinger

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

January, 1994

Abstract

Although several large sized 0-1 Knapsack Problems (KP) may be easily solved, it is often the case that most of the computational effort is used for preprocessing, i.e. sorting and reduction. In order to avoid this problem it has been proposed to solve the so-called core of the problem: A Knapsack Problem defined on a small subset of the variables. But the exact core cannot be identified without solving KP, so till now approximated core sizes had to be used.

In this paper we present an algorithm for KP which has the property that the obtained core size is minimal, and that the computational effort for sorting and reduction also is minimal. The algorithm is based on a dynamic programming approach, where the core size is extended by need, and the sorting and reduction is performed in a similar "lazy" way. The breadth-first search of the dynamic programming approach implies that all possible variations of the solution vector have been tested, before a new variable is introduced to the core. As a consequence, no unnecessary sorting and reduction is performed thus ensuring minimality.

Computational experiments are presented for several commonly occurring types of data instances. Experience from these tests indicate that the presented approach outperforms any known algorithm for KP.

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

Introduction

Given n *items* to pack in some knapsack of *capacity* c. Each item j has a *profit* p_j and *weight* w_j, and we wish to maximize the profit sum of the included items without having the weight sum to exceed c. More formally we define the *0-1 Knapsack Problem* (*KP*) by

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

*Technical Report 94/23, DIKU, University of Copenhagen, Denmark.

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

where all coefficients are positive integers. Without loss of generality we may assume that $w_j \le c$ for $j = 1,\ldots,n$ so each item fits into the knapsack, and that $\sum_{j=1}^{n} w_j > c$ to ensure a nontrivial problem. If we relax the integrality constraint $x_j \in \{0,1\}$ in (1) to the linear constraint $0 \le x_j \le 1$, we obtain the *Linear Knapsack Problem* (*LKP*).

Many industrial problems can be formulated as Knapsack Problems: Cargo loading, cutting stock, project selection, and budget control to mention a few examples. Many combinatorial problems can be reduced to KP, and the problem arises also as a subproblem in several algorithms of integer linear programming. KP is NP-hard (see Garey and Johnson 1979), but it can be solved in pseudo-polynomial time by dynamic programming (Papadimitriou 1981).

In the middle of the 1970ies several good algorithms for KP were developed (Horowitz and Sahni 1974, Nauss 1976, Martello and Toth 1977). The starting point of each of these algorithms was to order the variables according to nonincreasing profit-to-weight (p_j/w_j) ratio, which was the basis for solving LKP. From this solution appropriate upper and lower bounds were derived, making it possible to apply some logical tests to fix as many variables as possible at their optimal value. Finally the KP in the remaining variables was solved by branch-and-bound techniques.

However computational experience showed that the preprocessing (i.e. sorting and problem reduction) usually constituted the lion's share of the computational effort required to solve KP. Balas and Zemel (1980) avoided this problem by focusing on a small subset of the items — the so-called core — where there was a large probability for finding an optimal solution. The exact core consists of those variables whose profit-to-weight ratio falls between the maximum and minimum p_j/w_j ratio for which x_j in an optimal solution to KP has a different value from that in an optimal solution to LKP. Since the determination of the exact core would require the solving of KP, Balas and Zemel (1980) proposed to use an approximate core, which could be found through a partitioning technique of complexity $O(n)$. A complete sorting of the variables would require $O(n \log n)$. Martello and Toth (1988) modified the partitioning algorithm to satisfy some given requirements on the core size. But still the expected core size was a pure guess. Although the exact core cannot be determined *before* KP is solved, Pisinger (1994a) observed that the core can be determined *while* KP is solved, by simply adding new items to the core by need. However Pisinger used a depth-first branch-and-bound algorithm for the solution of KP, which had the disadvantage that an unpromising branch sometimes was followed to completion — thus forcing a further extension of the core, although an optimal solution could be found within the current core.

In this paper we avoid that disadvantage by using a breadth-first dynamic programming algorithm to enumerate the core before it is extended. This ensures that all possible variations of the solution vector have been tested before a new variable is introduced, thus ensuring the minimality.

This paper is organized the following way: First, Section 1 brings some basic definitions, while Section 2 shows how an initial core may be derived through a modified QUICKSORT algorithm. Next, Section 3 gives a description of the dynamic programming algorithm and Section 4 shows how the core may be expanded by need. The following sections show how we use some logical tests to fix as many variables as possible at their optimal value. Finally Section 7 brings computational experience.

A first version of this paper was presented at the NOAS'93 Conference (Pisinger 1993). Similar results as presented in this paper have recently been obtained for the Multiple-Choice Knapsack Problem and the Bounded Knapsack Problem (Pisinger 1994c, 1994d).

1 Definitions and main algorithm

The Linear Knapsack Problem may be solved by simply ordering the items according to nonincreasing *efficiencies* $e_j = p_j/w_j$ and then use the *greedy algorithm* for filling the knapsack: Include items $j = 1, 2, \ldots$ as long as

$$\sum_{i=1}^{j} w_i \leq c. \tag{2}$$

The first item b which cannot be included in the knapsack is denoted the *break item* and an optimal solution to LKP is given by including all items $1, \ldots, b-1$ and a fraction of item b to the knapsack (Dantzig 1957). Thus $x_j = 1$ for $j = 1, \ldots, b-1$ and $x_j = 0$ for $j = b+1, \ldots, n$ while x_b is given by

$$x_b = \frac{c - \sum_{i=1}^{b-1} w_i}{w_b}. \tag{3}$$

The corresponding pure integer solution $x' = \{x_1', \ldots, x_n'\}$ is known as the *break solution* and the variables are given by $x_j' = 1$ for $j = 1, \ldots, b-1$ and $x_j' = 0$ for $j = b, \ldots, n$.

Balas and Zemel (1980) observed that an *optimal solution* x^* to KP generally corresponds to the break solution x' except some few variables who have been changed. Figure 1 illustrates this property by measuring how often a variable is set to $x_j^* = 0$ for $j < b$ and $x_j^* = 1$ for $j \geq b$ in the optimal solution to KP. The figure is a result of solving 1000 randomly generated data instances of size $n = 1000$, with the capacity c chosen such that $b = 500$ for all instances. The figure shows that the frequency decreases steeply with j's distance from b. In average only 3.4 variables differ from the break solution per data instance.

This observation motivates considering only a small amount of the items around b in the solution process. In our definition a *core* is simply an interval $[s, t]$, $s \leq b \leq t$ of variables satisfying the *weak sort criteria*

$$\begin{array}{ll} \forall i, j \in [s, t] & : \; i < j \Rightarrow e_i \geq e_j, \\ \forall j \in [1, s-1] : & e_j \geq e_s, \\ \forall j \in [t+1, n] : & e_j \leq e_t. \end{array} \tag{4}$$

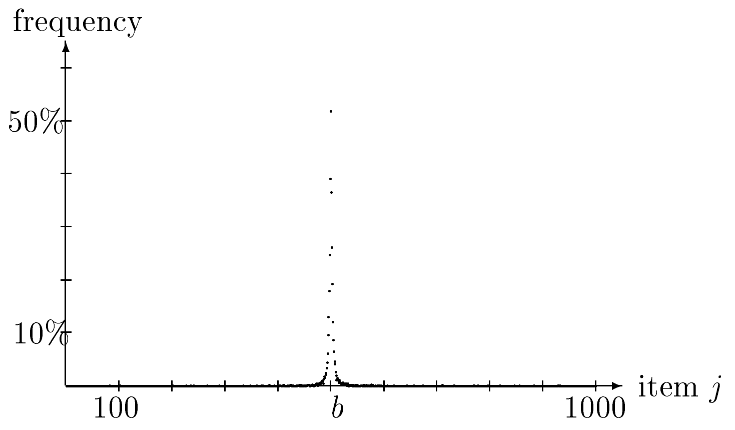

Figure 1: Frequency of items j where the optimal solution x_j^* differ from the break solution x'_j. Average of 1000 instances.

As long as we only consider variables in the core, this ordering is just as satisfying as a complete ordering of the items. Starting with $[s,t] = [b,b]$ we will enumerate all partial vectors in the core and alternately expand the core to the left and to the right. The set of partial vectors at any step is given by

$$X_{s,t} = \Big\{ \ (x_s, \ldots, x_t) \mid x_s, \ldots, x_t \in \{0,1\} \ \Big\}, \tag{5}$$

but we will use some dominance and upper bound tests to fathom unpromising branches. Clearly the enumeration of the core has time complexity $O(2^{t-s})$, so any effort possible should be used to avoid inclusion of new variables to the core. We have chosen to use an upper bound test for this purpose, fathoming a variable if the corresponding upper bound does not exceed the current best solution (*lower bound*) z. A *strong upper bound* is used for this test (Pisinger 1994b). Since all coefficients are integers we may fathom the variable if the strong upper bound is less than $z+1$ as no objective values between z and $z+1$ can occur.

The sorting of the variables according to nonincreasing efficiencies is much less complex, having an average execution time of $O(n \log n)$. Still, quite a lot of computational effort may be saved by using an upper bound test with a cheaply evaluated bound, to fathom unpromising variables. For this purpose we have chosen the bound by Dembo and Hammer (1980) which can be evaluated in constant time, giving the reduction algorithm a complexity of $O(n)$. This bound will be denoted the *weak upper bound*.

Since all these reductions are done by need, we use the following intervals to denote enumerated, sorted and reduced intervals (cf. Figure 2):

- $[s'', t'']$ is the interval of variables which have been tested by an upper bound test to

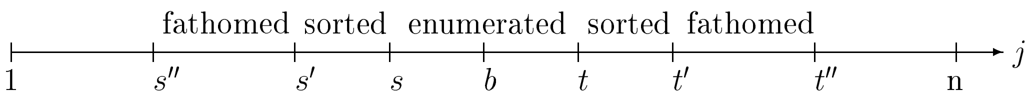

Figure 2: The intervals $[s,t]$, $[s',t']$ and $[s'',t'']$.

decide whether a change in the corresponding solution variable x_j may lead to an improved solution.

- $[s', t']$ is the subset of variables in $[s'', t'']$ which have weak upper bound larger than the current lower bound z. The variables in $[s', t']$ are ordered according to nonincreasing efficiencies.
- $[s, t]$ determines the core, i.e. variables which have been enumerated to $X_{s,t}$.

We have $[s, t] \subseteq [s', t'] \subseteq [s'', t'']$, and note that the intervals $[s'', s' - 1]$ and $[t' + 1, t'']$ contains fathomed variables. This lead us to the following main algorithm:

Algorithm 1

```
**procedure** minknap($n, c, p_1 \dots p_n, w_1 \dots w_n, x_1 \dots x_n$);
*Find break item b through partial sorting.*
$[s,t] := [b,b]$; $[s',t'] := [b,b]$; $[s'',t''] := [b,b]$;
$z := 0$; $X_{s,t} := \{(0),(1)\}$;
reduceset($X_{s,t}$);
**while** ($X_{s,t} \neq \emptyset$) **do**
  **if** ($s - 1 \geq s'$) **then**
    **if** strong upper bound($s-1$) $\geq z + 1$ **then** $X_{s-1,t} :=$ add($X_{s,t}, s-1$); **fi**;
    $s := s - 1$;
  **fi**;
  reduceset($X_{s,t}$);
  **if** ($t + 1 \leq t'$) **then**
    **if** strong upper bound($t+1$) $\geq z + 1$ **then** $X_{s,t+1} :=$ add($X_{s,t}, t+1$); **fi**;
    $t := t + 1$;
  **fi**;
  reduceset($X_{s,t}$);
**elihw**;
definesolution;
```

The first step of the algorithm is to find the break item b through partial sorting, which also returns some intervals $H = \{H_1, \dots, H_h\}$ and $L = \{L_1, \dots, L_l\}$ of partially ordered variables, where variables in H_j have higher efficiency than e_b while variables in L_j have lower efficiency. This algorithm will be explained further in Section 2. After some initializations, we repeatedly include a new variable $s - 1$ or $t + 1$ to the core, thus obtaining a larger set $X_{s,t}$. This is done by the function ADD which will be described in Section 3. After each inclusion of a new variable in $X_{s,t}$ we use some upper bound tests, to fathom unpromising vectors $\overline{x}_j \in X_{s,t}$. The upper bounds involved are obtained through linear relaxations of variables $s - 1$ and $t + 1$, and since these variables may fall outside the sorted set $[s', t']$, we will expand the core in such cases. This is all done in procedure REDUCESET, which will be explained in the second part of Section 3. We use the strong upper bound test, to determine whether a new variable $s - 1$ or $t + 1$ should be added to the core. The strong upper bound is described in Section 5, where we also give a thumb-rule for when it is worth evaluating the bound. Finally the solution vector x^* is defined: Since it seems awkward to represent elements in $X_{s,t}$ by complete vectors, some packing of the information is necessary. This will be discussed in Section 6.

We claim that Algorithm 1 solves KP to optimality with a minimal core and with minimal effort for sorting and reduction. More precisely we have

Definition 1 Given a core $[s, t]$ and the corresponding set of states $X_{s,t}$. We say that the core problem has been *solved to optimality* if one (or both) of the following cases occur:

- 1 $X_{s,t} = \emptyset$ meaning that all states were fathomed due to an upper bound test.
- 2 All items $j \in [1, s-1]$ could be fixed at $x_j = 1$ and all items $j \in [t+1, n]$ could be fixed at $x_j = 0$.

Definition 2 KP has been solved with a *minimal core* if the following invariant holds: A variable $s-1$ (resp. $t+1$) is only introduced to the core $[s, t]$ if $X_{s,t}$ could not be solved to optimality, and the inclusion of $s-1$ (resp. $t+1$) introduces at least one vector $\overline{x}_j$ to $X_{s-1,t}$ (resp. $X_{s,t+1}$) which has upper bound greater than z.

The definition states, that a variable $s-1$ (resp. $t+1$) should be introduced to the core only if it cannot be avoided by any upper bound test, and if all variables of lower (resp. higher) efficiency have been considered. The definition ensures that if KP has been solved to optimality with a minimal core $[s, t]$, no smaller *subset* core $[\tilde{s}, \tilde{t}] \subset [s, t]$ exists. Anyway a smaller *sized* core $[\tilde{s}, \tilde{t}]$ may exist if $\tilde{s} < s$ and $\tilde{t} < t$ (resp. $\tilde{s} > s$ and $\tilde{t} > t$) but according to our definition such cores are not comparable. Algorithm 1 finds the minimal core (of several possible) which is symmetric around b.

Definition 3 The sorting effort has been *minimal* if an interval $[f, l]$ is sorted only when

- $X_{s,t}$ could not be solved to optimality.
- All variables in $[f, l]$ have (weak) upper bound larger than z, i.e. all variables seems promising.
- ($s = s'$ and $[f, l] = H_h$) or ($t = t'$ and $[f, l] = L_l$).

The last expression states, that the core should not be extended until $[s, t]$ reaches the border $[s', t']$, and in such case $[f, l]$ should be chosen as the partially sorted interval H_h or L_l closest to $[s, t]$.

Definition 4 The reduction effort has been *minimal* if

- The weak upper bound is applied only when a new interval from H or L must be sorted according to the rule in definition 3.
- The strong upper bound is applied only when a new variable must be introduced to the core according to the rule in definition 2.

The following sections will show, that MINKNAP has all of these properties.

2 A partitioning algorithm for finding the break item

A complete sorting of the variables according to nonincreasing efficiencies e_j may be done in $O(n \log n)$ by a sorting algorithm like QUICKSORT (Hoare 1962). The QUICKSORT algorithm repeatedly picks a middle value λ from the interval $I = [f, l]$, and partition the interval in two parts $[f, i-1]$ and $[i, l]$, so that

$$e_j \geq \lambda, \quad j \in [f, i-1], \tag{6}$$

$$e_j \leq \lambda, \quad j \in [i, l]. \tag{7}$$

Initially $[f, l]$ is chosen as $[1, n]$, and the interval is then repeatedly partitioned in smaller parts, till a complete sorting has been achieved. Since we only need the partial ordering (4) for an initial core $[s, t] = [b, b]$, Pisinger (1994a) noted that several of these iterations may be discarded. Any interval $[f, i-1]$ in (6) with $\sum_{j=1}^{i-1} w_j \leq c$ may be discarded since b cannot be in the interval. Similarly an interval $[i, l]$ in (7) may be discarded if $\sum_{j=1}^{i-1} w_j > c$. The discarded intervals represent a partial ordering of $[1, n]$, and should thus be saved for future use. At any stage we have the invariants $W = \sum_{j=1}^{i-1} w_j$ and $V = \sum_{j=1}^{f-1} w_j$, thus getting the following modified version of QUICKSORT:

```
Algorithm 2
procedure partsort($f, l, V$);  {$[f, l]$ interval, $V = \sum_{j=1}^{f-1} w_j$}
$d := l - f + 1$;  {d is the size of the interval}
if ($d > 1$) then
   $m := \lfloor f + d/2 \rfloor$;  {m is the middle of the interval}
   Swap items $f, m, l$ so that $e_f \geq e_m \geq e_l$.
fi;
if ($d \leq 3$) then
   $\phi := f$;  $\psi := l$;  {return found interval $[\phi, \psi]$}
else
   $\lambda :=$ choosemedian($f, l$);  {$\lambda$ is the median of a subset of $[f, l]$}
   $i := f$;  $j := l$;  $W := V$;
   repeat  {partition $[f, l]$ in two intervals}
      repeat $W = W + w_i$;  $i := i + 1$; until ($e_i \leq \lambda$);
      repeat                 $j := j - 1$; until ($e_j \geq \lambda$);
      if ($i < j$) then swap($i, j$); fi;
   until ($i > j$);
    {now $e_k \geq \lambda$ for $k \in [f, j]$, and $e_k \leq \lambda$ for $k \in [i, l]$}
   if ($W > c$) then $H := H \cup \{[i, l]\}$;  partsort($f, i-1, V$);
                else $L := L \cup \{[f, i-1]\}$;  partsort($i, l, W$); fi;
fi;
```

In the above algorithm, SWAP(i, j) exchanges the items corresponding to indices i and j.

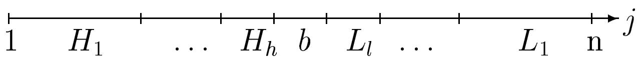

Figure 3: The lists H and L.

The PARTSORT algorithm is a common variant of the QUICKSORT algorithm with the modification that only intervals containing the break item are partitioned further. The algorithm terminates when the current interval contains at most three items $[\phi, \psi]$ with $\phi \leq b \leq \psi$. The sketched algorithm corresponds to Pisinger (1994a), except that we choose λ as the exact median of $\sqrt{d}$ randomly chosen items in $[f, l]$ for large intervals ($d \geq 100$). For small intervals we choose λ as e_m. This approach saves up to 30% of the computing time compared to Pisinger (1994a).

Note that all discarded intervals are added to the lists $H = \{H_1, \ldots, H_h\}$ and $L = \{L_1, \ldots, L_l\}$. Upon termination these intervals are ordered as indicated in Figure 3, and we have

$$\forall i \in H_k \; \forall j \in H_{k+1}: \; e_i \geq e_j, \quad k = 1, \ldots, h-1, \tag{8}$$
$$\forall i \in L_k \; \forall j \in L_{k+1}: \; e_i \geq e_j, \quad k = 1, \ldots, l-1. \tag{9}$$

3 A dynamic programming algorithm

In Section 1 we defined the set of all partial vectors in the core $[s, t]$ by

$$X_{s,t} = \{ \; (x_s, \ldots, x_t) \mid x_s, \ldots, x_t \in \{0, 1\} \; \}. \tag{10}$$

It is convenient to represent each partial vector $\overline{x}_i \in X_{s,t}$ by a *state* (π_i, μ_i, v_i) where π_i and μ_i are the profit and weight sums of the extended vector $\overline{x}_i$, given by $\overline{x}_{ij} = 1$ for $j = 1, \ldots, s-1$, and $\overline{x}_{ij} = 0$ for $j = t+1, \ldots, n$. Thus

$$\pi_i \;=\; \sum_{j=1}^{s-1} p_j + \sum_{j=s}^{t} p_j \overline{x}_{ij}, \tag{11}$$
$$\mu_i \;=\; \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j \overline{x}_{ij}. \tag{12}$$

The vector v_i is a (not necessarily complete) representation of the binary vector $\overline{x}_i$. According to the principle of optimality (Ibaraki 1987) we may fathom some of these states:

Definition 5 Given two states (π_i, μ_i, v_i) and (π_j, μ_j, v_j). The state i is said to *dominate* the state j if $\pi_i \geq \pi_j$ and $\mu_i \leq \mu_j$.

Proposition 1 If a state i dominates another state j we may fathom the dominated state j.

Proof Let $\overline{x}_i$ and $\overline{x}_j$ be the binary vectors corresponding to states i and j. Assume that an optimal solution x^* contains $\overline{x}_j$ as a sub-vector. Then an equally good or better solution may be obtained by setting $x^*_k = \overline{x}_{ik}$ for $k = s, \ldots, t$. $\Box$

We will keep the set $X_{s,t} = \{(\pi_1, \mu_1, v_1), \ldots, (\pi_m, \mu_m, v_m)\}$ ordered according to increasing profit and weight sums ($\pi_i < \pi_{i+1}$ and $\mu_i < \mu_{i+1}$) in order to easily fathom dominated states. The following algorithm shows how an ordered set $X_{s,t}$ may be extended with a new variable x_ℓ, obtaining a new ordered set with dominated states removed. Note, that if $\ell < b$ then (p_ℓ, w_ℓ) should be subtracted from the states, while if $\ell \geq b$ they should be added to the states.

The problem is to merge two ordered sets X and $X + \ell$ (the set X with variable ℓ added/ subtracted to/from all states) ensuring that the product set is also ordered and dominated states are removed. We will let i indicate the current considered state in X while j indicates the current state in $X + \ell$. The last state in the product set X' is denoted k.

The algorithm repeatedly choose the least weighted of the two states $i \in X$ and $j \in X + \ell$ in order to ensure the ordering of X'. Dominated states are deleted either by skipping the state i (resp. j) or by overriding the state k.

Algorithm 3

```
function add($X$, $\ell$);
 {input: $X = \{(\pi_1, \mu_1, v_1), \ldots, (\pi_m, \mu_m, v_m)\}$,      }
 {        $\ell$ = new variable to be included in core.}
 {output: $X' = \{(\pi'_1, \mu'_1, v'_1), \ldots, (\pi'_{m'}, \mu'_{m'}, v'_{m'})\}$. }
if ($\ell < b$) then $\tilde{p} = -p_\ell$; $\tilde{w} = -w_\ell$; else $\tilde{p} = p_\ell$; $\tilde{w} = w_\ell$; fi;  {subtract or add item $\ell$}
$i := 1$; $j := 1$; $k := 1$;  {initialize}
$(\mu_{m+1}, \pi_{m+1}, v_{m+1}) := (\mu_m + w_\ell + 1, \pi_m + p_\ell + 1, \emptyset)$;  {add state for correct termination}
if ($\mu_i \leq \mu_j + \tilde{w}$) then $(\mu'_k, \pi'_k, v'_k) := (\mu_i, \pi_i, v_i)$; $i := 2$;  {add state for correct start}
                   else $(\mu'_k, \pi'_k, v'_k) := (\mu_j + \tilde{w}, \pi_j + \tilde{p}, v_j \cup \{\ell\})$; $j := 2$; fi;
repeat
   if ($\mu_i \leq \mu_j + \tilde{w}$) then  {choose smallest weight to ensure ordering.}
      if $(\mu_i, \pi_i, v_i)$ is not dominated by $(\mu'_k, \pi'_k, v'_k)$ then
         if $(\mu_i, \pi_i, v_i)$ does not dominate $(\mu'_k, \pi'_k, v'_k)$ then k := k + 1; fi;
         $(\mu'_k, \pi'_k, v'_k) := (\mu_i, \pi_i, v_i)$;
      fi;  $i := i + 1$;
   else
      if $(\mu_j + \tilde{w}, \pi_j + \tilde{p}, v_j)$ is not dominated by $(\mu'_k, \pi'_k, v'_k)$ then
         if $(\mu_j + \tilde{w}, \pi_j + \tilde{p}, v_j)$ does not dominate $(\mu'_k, \pi'_k, v'_k)$ then k := k + 1; fi;
         $(\mu'_k, \pi'_k, v'_k) := (\mu_j + \tilde{w}, \pi_j + \tilde{p}, v_j \cup \{\ell\})$;
      fi;  $j := j + 1$;
   fi;
until ($i = m + 1$) and ($j = m + 1$);
$m' = k$; return $X'$;
```

Algorithm 3 allows us to generate all undominated states in $X_{s,t}$ through dynamic programming, but the technique may be improved by deleting unpromising states after each iteration as indicated by procedure REDUCESET in Algorithm 1. Unpromising states have an upper bound, which does not exceed the current best solution z. For a state i given by (π_i, μ_i, v_i) we use the upper bound

$$u(i) \;=\; \begin{cases} u_1(i) \;=\; \pi_i + \dfrac{(c-\mu_i)\cdot p_{t+1}}{w_{t+1}} & \text{if} \quad \mu_i \le c, \\ \\ u_2(i) \;=\; \pi_i + \dfrac{(c-\mu_i)\cdot p_{s-1}}{w_{s-1}} & \text{if} \quad \mu_i > c, \end{cases} \tag{13}$$

which has been obtained by relaxing the constraints on x_{s-1} and x_{t+1} to $x_{s-1} \ge 0$ and $x_{t+1} \ge 0$. Since $s-1$ or $t+1$ 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 $s'' = 1$, all intervals H have been considered, so the core cannot be expanded further to this side. Choose $p_{s-1} = \infty$ and $w_{s-1} = 1$. Similarly if $t'' = n$ 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 choose an appropriate interval H_h or L_l, and reduce concerned variables through a (weak) upper bound test. This test will be described in the next section.

- If all variables in H_h or L_l have been fathomed through the upper bound test, we can choose any variable from the concerned interval as $s-1$ resp. $t+1$ in (13). Otherwise the remaining variables are ordered according to nonincreasing efficiencies and added to the set of sorted variables $[s', t']$. Equation (13) may be used directly.

The bound (13) may also be used for deriving a global upper bound on KP. Since any optimal solution must follow a branch in $X_{s,t}$, the global upper bound corresponds to the upper bound of the most feasible branch in $X_{s,t}$. Therefore a global upper bound on KP is given by

$$u_{\text{KP}} \;=\; \max_{i \in X} u(i). \tag{14}$$

Since the efficiency of variable $t+1$ will be decreasing during the solution process, and the efficiency of $s-1$ will be increasing, u_{KP} will become more and more tight during the solution process. For $(s,t) = (1,n)$ we get $u_{\text{KP}} = z$ for the optimal solution z.

4 Weak reduction

Assume that the solution vector x corresponding to the current lower bound z has been saved. If an upper bound on KP with the additional constraint $x_j = 0$, $j < b$ (resp. $x_j = 1$, $j \ge b$) is less than $z+1$ we may conclude that the branch $x_j = 0$ (resp. $x_j = 1$) will never lead to an improved solution, and can thus fix x_j at 1 (resp. 0).

Let u_j^0 (resp. u_j^1) be an upper bound on KP with the additional constraint $x_j = 0$ (resp. $x_j = 1$), then the reduction scheme is

$$\begin{aligned} u_j^0 < z+1 &\;\Rightarrow\; x_j = 1, \qquad && j = 1,\dots,b-1, \\ u_j^1 < z+1 &\;\Rightarrow\; x_j = 0, \qquad && j = b,\dots,n. \end{aligned} \tag{15}$$

Different upper bounds are presented in Ingargiola and Korsh (1973), Dembo and Hammer (1980), Fayard and Plateau (1982), and Martello and Toth (1988). For our purpose we have chosen the bounds by Dembo and Hammer:

$$\begin{aligned} u_j^0 &= \overline{p} - p_j + \frac{(r + w_j) \cdot p_b}{w_b}, \qquad && j = 1,\dots,b-1, \\ u_j^1 &= \overline{p} + p_j + \frac{(r - w_j) \cdot p_b}{w_b}, \qquad && j = b,\dots,n, \end{aligned} \tag{16}$$

where $\overline{p}$ is the profit sum $\sum_{i=1}^{b-1} p_i$ and r is the residual capacity $c - \sum_{i=1}^{b-1} w_i$. This bound is not so tight, but it only demands the partial ordering (4) and can be evaluated in constant time, thus making it suitable for an upper bound test before sorting the next interval. Since the reduction is performed dynamically throughout the solution process, and not like in traditional algorithms as a part of the preprocessing, we may expect that a better solution (lower bound) is found during the enumeration, thus compensating for the weakness. We obtain the following sketch of the reduction algorithm:

Algorithm 4

```
procedure reduce($f, l$);
if ($l < b$) then  {$[f, l]$ is left of $b$}
  for $j := f$ to $l$ do
    $u_j^0 := \overline{p} - p_j + \frac{(r + w_j) \cdot p_b}{w_b}$;  {Upper bound for branch $x_j = 0$}
    if ($u_j^0 < z + 1$) then $x_j$ is fixed to 1: Swap $j$ to $[s'', s' - 1]$, extend the interval.
                         else $x_j$ is free: Swap $j$ to $[s', s - 1]$, extend the interval. fi;
  rof;
  Sort the new variables in $[s', s - 1]$.
else  {$[f, l]$ is right of $b$}
  for $j := f$ to $l$ do
    $u_j^1 := \overline{p} + p_j + \frac{(r - w_j) \cdot p_b}{w_b}$;  {Upper bound for branch $x_j = 1$}
    if ($u_j^1 < z + 1$) then $x_j$ is fixed to 0: Swap $j$ to $[t' + 1, t'']$, extend the interval.
                         else $x_j$ is free: Swap $j$ to $[t + 1, t']$, extend the interval. fi;
  rof;
  Sort the new variables in $[t + 1, t']$.
fi;
```

The algorithm receives an interval $[f, l]$ given by H_h or L_l and tests whether the corresponding variables may be fixed at their optimal value. The two while-loops are performed in linear time, implying that each reduction of an interval $[f, l]$ is performed in $O(l - f + 1)$.

5 Strong upper bound

Since the addition of a new item ℓ to the core is computationally very expensive, a strong upper bound test should be used for checking whether the inclusion seems promising. The ultimate check is to determine whether any states in $X + \ell$ will pass the reduction (13). No stronger bound can be constructed, since if the inclusion of ℓ to the core implies that a new promising branch is introduced to $X_{s,t}$, clearly ℓ has to be included.

A state i in $X + \ell$ is the sum $(\pi_i + \tilde{p}, \mu_i + \tilde{w}, v_i \cup \{\ell\})$ of the corresponding state i in X and the variable ℓ (added or subtracted), so an upper bound for the state is

$$\tilde{u}(i) = \begin{cases} \tilde{u}_1(i) = (\pi_i + \tilde{p}) + \dfrac{(c - (\mu_i + \tilde{w})) \cdot p_{t+1}}{w_{t+1}} & \text{if} \quad \mu_i + \tilde{w} \le c, \\ \\ \tilde{u}_2(i) = (\pi_i + \tilde{p}) + \dfrac{(c - (\mu_i + \tilde{w})) \cdot p_{s-1}}{w_{s-1}} & \text{if} \quad \mu_i + \tilde{w} > c, \end{cases} \tag{17}$$

where we simply have used (13) for the set $X + \ell$. An upper bound for the inclusion of variable ℓ into the core is thus given by

$$\tilde{u}_\ell = \max_{i \in X} \tilde{u}(i). \tag{18}$$

This bound may be recognized as the π-bound presented in Pisinger (1994b). Note that the functions $\tilde{u}_1$ and $\tilde{u}_2$ may be written

$$\begin{aligned} \tilde{u}_1(i) &= u_1(i) + \tilde{p} - \frac{\tilde{w} \cdot p_{t+1}}{w_{t+1}}, \\ \tilde{u}_2(i) &= u_2(i) + \tilde{p} - \frac{\tilde{w} \cdot p_{s-1}}{w_{s-1}}, \end{aligned} \tag{19}$$

where $u_1(i)$ and $u_2(i)$ are the upper bounds of state $i \in X$ as given by (13).

Proposition 2 The bound $\tilde{u}_\ell$ exceeds the lower bound z if and only if a state in $X + \ell$ will pass the reduction (13).

Proof Assume that $\tilde{u}_\ell = a$ with $a \ge z + 1$. Since $\tilde{u}_\ell$ is the maximum of bounds (17), choose the state $i \in X$ which satisfies $\tilde{u}(i) = a$. The state $j \in X + \ell$, which is obtained by adding variable ℓ to i, will also have upper bound $a \ge z + 1$, meaning that it passes the fathoming test (13).

Contrary, if a state j in $X + \ell$ passes the fathoming test (13), its upper bound must exceed z, thus forcing the bound $\tilde{u}_\ell$ to exceed z. $\Box$

Unfortunately the complexity of determining $\tilde{u}_\ell$ is $O(m)$, where m is the number of states in $X_{s,t}$, meaning that the computational effort for deriving the strong upper bound corresponds to the computational effort of including variable ℓ to the core. Therefore we will only evaluate the bound if there is a good chance of fathoming the concerned variable.

Note that $(\tilde{p}, \tilde{w})$ corresponds to (p_{t+1}, w_{t+1}) or $(-p_{s-1}, -w_{s-1})$ since we are testing the inclusion of variable $t + 1$ or $s - 1$. So in the first case we have $\tilde{u}_1(i) = u_1(i)$ and in the second $\tilde{u}_2(i) = u_2(i)$, which may be verified by inserting $\tilde{p}, \tilde{w}$ in (19). Since $u_1(i) \geq z+1$ and $u_2(i) \geq z+1$ due to the fathoming test(13), the bound $\tilde{u}_\ell$ can only be less than $z+1$ if the sets $\{i \in X_{s,t} \mid \mu_i + w_{t+1} \leq c\}$, respectively $\{i \in X_{s,t} \mid \mu_i - w_{s-1} > c\}$ are empty. We have shown the following proposition:

Proposition 3 If $\ell = t+1$ and $\mu_1 \leq c - w_{t+1}$ then $\tilde{u}_\ell \geq z+1$. If $\ell = s-1$ and $\mu_m > c + w_{s-1}$ then $\tilde{u}_\ell \geq z+1$.

Computational experience show, that if the criteria in Proposition 3 do not hold, then $\tilde{u}_\ell < z+1$ in more than 70% of the cases. Therefore the evaluation of $\tilde{u}_\ell$ in such cases is worth the effort, since we generally may fathom the concerned variable.

6 Finding the solution vector

According to Bellmans classical description of dynamic programming (Bellman 1957), the optimal solution vector x^* may be found by backtracking through the sets of states. But this technique means that all sets of states should be saved during the solution process. In the computational experience it is demonstrated that the number of states may be over 2 millions in each iteration. With $n = 100000$ as a measure for the number of iterations, we would need to store billions of states, so another strategy should be chosen. A promising approach seems to be, that only the last a changes in the solution vector are saved in the state variable v. In our implementation $a = 32$ was chosen.

Whenever an improved solution is found during the construction of $X_{s,t}$, we save the corresponding state (π, μ, v). After termination of the algorithm, the solution vector is tried reconstructed. First all variables x_i are set to the break solution $x_i{}'$. Then we run through the vector v in the following way:

```
**Algorithm 5**
**procedure** definesolution($\pi, \mu, v$);
 {$v = \{v_1, \dots, v_a\}$ *are indices to the last* $a$ *variables added to the state*}
**for** $i := 1$ **to** $a$ **do**
   $j := v_i$;  {$j$ *is the variable corresponding to* $v_i$}
   **if** $(j < b)$ **then** $x_j := 0$; $\pi := \pi + p_j$; $\mu := \mu + w_j$;
                **else** $x_j := 1$; $\pi := \pi - p_j$; $\mu := \mu - w_j$; **fi**;
**rof**;
```

If the backtracked profit and weight sums (π, μ) correspond to the profit and weight sums of the break solution (π', μ'), we know that the constructed vector is correct. Otherwise we solve a new KP, this time with capacity $c = \mu$, lower bound $z = \pi - 1$, and global upper bound $u = \pi$. The process is repeated till the solution vector x is completely defined.

This technique has proved very efficient, since generally only a few iterations are needed. A maximum of 16 iterations has been observed for large strongly correlated data instances, but otherwise one or two iterations suffice. In the worst case 12% of the solution time was used for reconstructing the solution vector. Compared to saving all states on an external device (which is hundred of times slower than the main memory) it is considerably more efficient to re-evaluate the states.

7 Computational experience

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 using the standard HP-UX C compiler with option -O (optimization).

We will consider how the algorithm behaves for different problem sizes, test instances, and data ranges. Four types of randomly generated data instances are considered as sketched below. Each type will be tested with *data range* $R_1 = 100$, $R_2 = 1000$, $R_3 = 10000$ for different problem sizes n =100, 300, 1000, 3000, 10000, 30000, 100000. The capacity c is chosen as $c = \frac{1}{2}\sum_{j=1}^{n} w_j$.

- *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 $p_j = w_j$.

For each problem type, size and range, we construct and solve 50 different data instances. The presented results are average values or extreme values. If a problem was not solved within 24 hours, this is indicated by a "—" in the tables. We will compare the computing times of MINKNAP to those of MT2 (Martello and Toth 1988). The code for MT2 was obtained from Martello and Toth (1990), in which MT2 also is compared to several algorithms for KP, showing that MT2 outperforms any of these. The MT2 code was compiled using the standard HP-UX FORTRAN compiler with option -O (optimization).

	UC			WC			SC			SS		
n	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
100	7	8	9	8	14	15	35	41	47	8	11	15
300	8	11	13	8	17	20	132	140	112	8	11	15
1000	7	16	17	8	16	25	428	384	386	7	11	15
3000	7	17	21	7	14	29	1105	1120	1270	8	12	14
10000	10	15	27	8	12	30	4119	3662	4003	8	12	15
30000	23	11	29	7	13	26	11938	12768	11738	7	11	15
100000	34	12	27	8	13	18	37144	43420	39649	7	11	15

Table I: Final core size (number of items). Average of 50 instances.

	UC			WC			SC			SS		
n	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
100	79	93	96	55	100	99	79	94	93	50	58	66
300	65	96	97	20	96	99	88	97	86	27	44	51
1000	12	88	95	5	68	98	94	85	92	20	33	28
3000	5	74	96	2	22	94	78	81	95	11	16	20
10000	2	31	90	0	8	78	93	83	86	4	3	14
30000	2	7	85	1	2	38	87	85	86	10	5	7
100000	0	1	54	0	0	12	89	91	90	0	1	5

Table II: Percentage of all items which have been tested by the weak upper bound. Average of 50 instances.

First Table I shows the average core size for solving KP to optimality. The core size is measured as the number of items in $X_{s,t}$, i.e. $(t - s + 1)$ minus the variables fathomed by the strong upper bound test. It is seen, that the core size is not of constant size, as claimed by Balas and Zemel (1980). For uncorrelated, weakly correlated and subset-sum data instances, the core size is very small, showing slight tendencies to grow with the data range. For strongly correlated data instances, the core size is large, since about half of the items must be considered in order to solve the problem.

Next Table II shows the average percentage of items, which need to be tested by the weak upper bound in order to solve KP. The presented numbers are determined as $(t'' - s'' + 1)/n$. We observe the interesting property, that large-sized uncorrelated, weakly correlated and subset-sum data instances generally can be solved without testing more than a few percent of the items. On the other hand strongly correlated data instances, and all small-sized data instances need a complete testing of the variables.

Table III gives the maximum number of states obtained in the solution process. For uncorrelated, weakly correlated and subset-sum data instances, less than 65.000 states are generated, indicating that the dynamic programming algorithm without complications may be applied on any computer. On the other hand strongly correlated data instances may involve more than 2.5 million states, which in our implementation takes up about 30Mb RAM. Papadimitriou (1981) showed that KP can be solved in pseudopolynomial time by dynamic programming, since the number of states at each step is limited by c.

	UC			WC			SC			SS		
n	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
100	0	0	0	0	0	0	1	8	78	0	6	63
300	0	0	0	0	1	1	1	14	148	1	5	53
1000	0	1	1	1	2	3	3	25	287	0	5	55
3000	0	1	1	0	4	7	4	45	425	0	5	60
10000	0	3	6	1	5	12	8	74	764	0	6	52
30000	2	4	9	0	11	32	15	130	1407	0	6	55
100000	4	4	19	0	27	65	37	250	2547	0	5	48

Table III: Largest set of states in dynamic programming (in thousands). Max of 50 instances.

	UC			WC			SC			SS		
n	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.43	0.00	0.00	0.06
300	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.18	1.63	0.00	0.00	0.06
1000	0.00	0.00	0.00	0.00	0.00	0.01	0.08	0.63	7.62	0.00	0.01	0.05
3000	0.00	0.01	0.01	0.00	0.01	0.02	0.24	2.28	42.65	0.01	0.01	0.06
10000	0.01	0.01	0.03	0.01	0.01	0.05	1.25	10.39	161.04	0.01	0.02	0.07
30000	0.03	0.03	0.07	0.03	0.03	0.08	3.15	42.78	491.31	0.05	0.04	0.10
100000	0.11	0.10	0.17	0.10	0.12	0.16	13.96	178.14	2208.82	0.11	0.12	0.20

Table IV: Total computing time in seconds (MINKNAP). Average of 50 instances.

	UC			WC			SC			SS		
n	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
100	0.00	0.00	0.00	0.00	0.00	0.01	2.78	2.68	21.16	0.00	0.00	0.01
300	0.00	0.00	0.00	0.00	0.00	0.01	—	—	—	0.00	0.00	0.02
1000	0.00	0.00	0.01	0.00	0.01	0.02	—	—	—	0.00	0.00	0.02
3000	0.00	0.01	0.02	0.00	0.01	0.07	—	—	—	0.00	0.00	0.02
10000	0.02	0.03	0.07	0.02	0.04	0.16	—	—	—	0.01	0.01	0.03
30000	2.47	0.07	0.20	0.05	0.61	0.26	—	—	—	0.03	0.03	0.05
100000	—	0.28	0.56	0.25	—	0.56	—	—	—	0.12	0.12	0.15

Table V: Total computing time in seconds (MT2). Average of 50 instances.

The table shows, that the actual number of states is far less than c, although strongly correlated data instances may involve up to 1% of the possible c states.

Finally Table IV shows the average computing time for each of the considered data instances. It is seen, that easy data instances may be solved in a fraction of a second even if the number of variables is 100000. Strongly correlated data instances demand considerably more computational effort, but are still solved within one hour of computation time on average.

This should be compared to the computing times for MT2 given in Table V. It is seen, that MT2 is not able to solve strongly correlated data instances of large size, and moreover the algorithm has some anomalous occurrences for large uncorrelated and weakly correlated data instaces. In these situations some instances could not be solved within 24 hours of computational time. As explained in Pisinger (1994a) this may be a consequence of the a priori determination of the core in MT2: If the guessed core is not appropriate, extensive branching is a consequence.

8 Conclusions

We have presented a complete algorithm for the exact solution of the 0-1 Knapsack Problem. The algorithm solves KP with a minimal core, since variables are introduced to the core only if the current core could not be solved to optimality, and the inclusion of the new variable introduces at least one promising state in the set $X_{s,t}$ (this is a consequence of Algorithm 1, the strong upper bound (18) and Proposition 2). Moreover the chosen strategy for core-expansion described in second part of Section 3 ensures that the effort for sorting and reduction has been minimal.

It is interesting to compare the obtained results to previous work: Balas and Zemel (1980) defined the core as the interval of sorted variables between first and last variable x_j^* which differ from the break solution x_j'. Even if such a core could be obtained a priori it would not guarantee that optimality could be proved by any upper bound, so the approach seems no good. Martello and Toth (1988) on the other hand chose a larger interval of variables around b for the core, namely n variables if $n \leq 100$, and $\sqrt{n}$ variables if $n > 100$. In Martello and Toth (1990) this core size is for unknown reasons changed to the double size. The presented minimal core sizes in Table I show that far smaller core sizes may be applied for uncorrelated and weakly correlated data instances, while strongly correlated data instances demand larger core sizes.

It should be mentioned, that the here stated minimality by no means guarantees that KP cannot be solved easier. Completely different approaches may show better results, and even similar types of algorithms may perform better if other upper bounds are applied. Note however that tighter upper bounds than the strong upper bound cannot be constructed, as discussed in the beginning of Section 5.

Apart from showing some minimal properties, we have derived a very efficient algorithm for the solution of KP. For uncorrelated, weakly correlated and subset-sum data instances it performs better and more stable than the so far best algorithm MT2 (Martello and Toth 1988). For strongly correlated data instances no algorithm has ever been able to solve instances of this size.

Finally we have seen that the global upper bound u_{KP} converge towards the lower bound z. This makes the MINKNAP algorithm well suited as an approximate algorithm, since for a given maximum tolerance, the algorithm may be terminated when the current solution differs less from the upper bound than demanded.

Appendix A: Additional computational results

In this section we bring the results of some additional computational experiments with MINKNAP.

	UC			WC			SC			SS		
$n \setminus R$	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
300	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.6	1.6	1.5	1.0	1.0	1.0
3000	1.0	1.0	1.0	1.0	1.0	1.2	2.3	2.3	2.5	1.0	1.0	1.0
10000	1.0	1.0	1.1	1.0	1.0	1.3	3.8	3.9	4.1	1.0	1.0	1.0
30000	1.2	1.0	1.0	1.0	1.0	1.1	4.7	6.6	6.5	1.0	1.0	1.0
100000	1.4	1.0	1.1	1.0	1.0	1.0	5.0	11.7	11.9	1.0	1.0	1.0

Table VI: Number of iterations used for obtaining the solution vector. Average of 50 instances.

	UC			WC			SC			SS		
$n \setminus R$	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
100	3.6	40.6	448.1	0.9	16.5	153.8	3.9	4.5	5.6	0.0	0.0	0.0
300	1.7	22.0	239.5	0.1	6.0	72.8	4.6	5.0	4.0	0.0	0.0	0.0
1000	0.3	8.6	88.9	0.0	1.8	27.5	4.9	4.2	4.1	0.0	0.0	0.0
3000	0.1	3.0	38.1	0.0	0.5	9.8	4.1	4.1	5.0	0.0	0.0	0.0
10000	0.0	0.8	14.8	0.0	0.0	3.1	4.9	4.4	4.8	0.0	0.0	0.0
30000	0.0	0.2	5.7	0.0	0.0	1.0	4.3	4.9	4.5	0.0	0.0	0.0
100000	0.0	0.0	1.7	0.0	0.0	0.2	4.1	4.9	4.6	0.0	0.0	0.0

Table VII: Gap Γ between LP-optimal solution and integer-optimal solution. Average of 50 instances.

	UC			WC			SC			SS		
$n \setminus R$	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
100	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.48	0.00	0.00	0.05
300	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.14	1.87	0.00	0.00	0.04
1000	0.00	0.00	0.00	0.00	0.00	0.01	0.07	0.64	8.62	0.00	0.01	0.05
3000	0.00	0.00	0.01	0.00	0.01	0.02	0.24	2.42	39.65	0.00	0.01	0.05
10000	0.00	0.00	0.01	0.00	0.01	0.03	1.13	10.77	152.45	0.01	0.01	0.04
30000	0.01	0.01	0.02	0.01	0.01	0.04	2.91	38.18	506.74	0.04	0.03	0.07
100000	0.03	0.01	0.04	0.00	0.05	0.05	13.95	156.27	2381.84	0.00	0.05	0.11

Table VIII: Standard deviation computational times MINKNAP. Average of 50 instances.

First, Table VI brings the number of iterations, which are needed to define the complete solution vector as described in Section 6. For all instances except the strongly correlated, slightly more than one iteration is needed on the average, meaning that the compact representation of the solution vector generally is sufficient. Only in a few cases, an additional iteration is needed, meaning that there is a minimal overhead for this part of the algorithm. For strongly correlated instances, however up to a dozen iterations are needed, but this still means that it is a negligible part of the solution time which is used for finding the solution vector.

Table VII shows the gap Γ between the LP-optimal and the integer-optimal solution. According to Balas and Zemel (1980), the hardness of a Knapsack Problem depends on the correlation of the data 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 with coefficients generated in a small range, as the gap grows with increasing data range. For strongly correlated instances, Γ is on the average constant around five. The increasing computational time for larger data ranges should merely be sought in the pseudopolynomial solution time of MINKNAP: Each time R is increased by a factor, the capacity c and thus the time-bound $O(nc)$ are increased by the same amount.

Finally Table VIII shows the standard deviation of the computational times of MINKNAP. Apart from the strongly correlated instances — which apparently have a very large variation in the running times — most of the variations are very small, and considerably less than one. This demonstrates that MINKNAP has a very stable behavior.

References

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

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

Dantzig, G.B., "Discrete Variable Extremum Problems," *Operations Research*, 5 (1957), 266-277.

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

Fayard, D. and G. Plateau, "An Algorithm for the Solution of the 0-1 Knapsack Problem," *Computing*, 28 (1982), 269-287.

Garey, M.R. and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

Hoare, C.A.R., "Quicksort," *Computer Journal*, 5, 1 (1962), 10-15.

Horowitz, E. and S. Sahni, "Computing partitions with applications to the Knapsack Problem," *Journal of ACM*, 21 (1974), 277-292.

Ibaraki, T., "Enumerative Approaches to Combinatorial Optimization - Part 2," *Annals of Operations Research*, 11 (1987).

Ingargiola, G.P. and J.F. Korsh, "A Reduction Algorithm for Zero-One Single Knapsack Problems," *Management Science*, 20 (1973), 460-463.

Martello, S. and P. Toth, "An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound algorithm," *Europan Journal of Operational Research*, 1 (1977), 169-175.

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, John Wiley & Sons Ltd., England, 1990.

Nauss, R. M., "An Efficient Algorithm for the 0-1 Knapsack Problem," *Management Science*, 23 (1976), 27-31.

Papadimitriou, C.H., “On the complexity of integer programming,” *Journal of ACM*, 28 (1981), 765-768.

Pisinger, D., “On the solution of 0-1 Knapsack Problems with minimal preprocessing,” *Proceedings NOAS'93,* Trondheim, Norway, June 11-12. (1993).

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

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

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

Pisinger, D., “A minimal algorithm for the Bounded Knapsack Problem,” *DIKU, University of Copenhagen, Denmark,* Report 94/27 (1994d).

Figures for reproduction

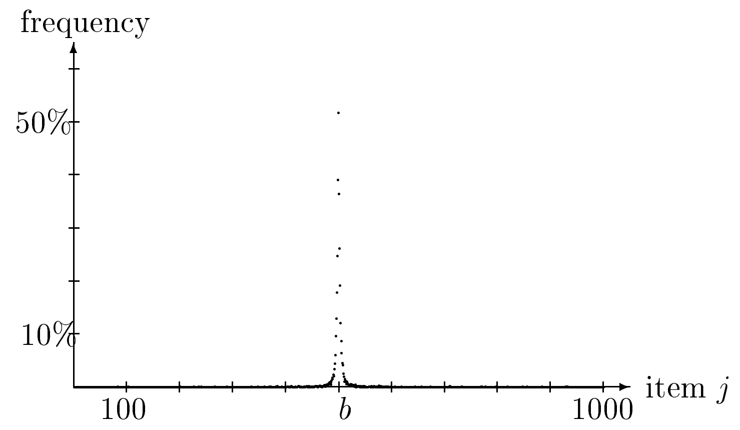

Figure 1: Frequency of items j where the optimal solution x_j^* differ from the break solution x_j'. Average of 1000 instances.

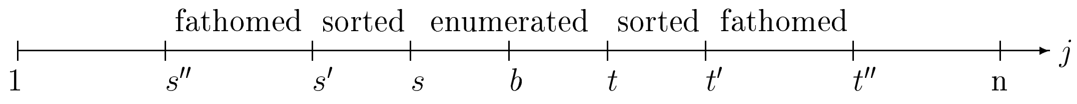

Figure 2: The intervals $[s, t]$, $[s', t']$ and $[s'', t'']$.

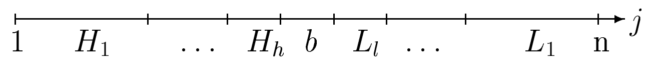

Figure 3: The lists H and L.