An expanding-core algorithm for the exact 0-1 Knapsack Problem.

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

December 3, 1993

Abstract

A new branch-and-bound algorithm for the exact solution of the 0-1 Knapsack Problem is presented. The algorithm is based on solving an "expanding core", which initially only contains the break item, but which is expanded each time the branch-and-bound algorithm reaches the border of the core. Computational experiments show that most data instances are optimally solved without sorting or preprocessing a great majority of the items. Detailed program sketches are provided, and computational experiments are reported, indicating that the algorithm presented not only is shorter, but also faster and more stable than any other algorithm hitherto proposed.

Keywords: Knapsack Problem, Branch-and-bound, Reduction.

1 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 without having the weight sum to exceed c. More formally we define the *0-1 Knapsack Problem* by

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

where all data are positive integers. In the following we will denote the Knapsack Problem by *KP*. Without loss of generality we may assume that $w_j \leq c$, for $j = 1, \ldots, n$ to ensure that each item considered fits into the knapsack, and that $\sum_{j=1}^{n} w_j > c$ to avoid trivial solutions.

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 KP arises also as a subproblem in several algorithms of integer linear programming.

KP is NP-hard (see Garey and Johnson (1979)) and it is therefore very unlikely that a polynomial time algorithm can be devised. Still KP can be solved in pseudo-polynomial time by dynamic programming (Papadimitriou (1981)). The problem has been intensively studied in the last decades due to its theoretical interest and its wide applicability. For recent surveys see Dudzinski and Walukiewicz (1987) and Martello and Toth (1990).

In the following we will assume, that the items are ordered according to nonincreasing *efficiencies* $e_j = p_j/w_j$, thus

$$e_i \geq e_j \quad \text{when} \quad i < j. \tag{2}$$

Although there may be several orderings satisfying (2), when some items have the same efficiency, we will assume, that one particular ordering has been chosen. To simplify notation, we will define the *profit sum* $\overline{p}_j$ and the *weight sum* $\overline{w}_j$ by:

$$\overline{p}_j = \sum_{i=1}^{j} p_i, \quad j = 0, \ldots, n, \tag{3}$$

$$\overline{w}_j = \sum_{i=1}^{j} w_i, \quad j = 0, \ldots, n. \tag{4}$$

Packing a knapsack in a *greedy way* means, to include items $j = 1, 2, \ldots,$ as long as $w_j \leq c - \overline{w}_{j-1}$, i.e. as far as the next item fits into the unused capacity of the knapsack. The first item which cannot be included in the knapsack is denoted the *break item* b (by some authors called the *critical item*). Thus the break item satisfies

$$\overline{w}_{b-1} \leq c < \overline{w}_b. \tag{5}$$

The *break solution* $x' = \{x'_1, \ldots, x'_n\}$ is the solution which occurs when setting $x'_j = 1$ for $j = 1, \ldots, b-1$ and $x'_j = 0$ for $j = b, \ldots, n$. The unused capacity of the break solution is called the *residual capacity* r, and is defined by

$$r = c - \overline{w}_{b-1}. \tag{6}$$

By linear relaxation Dantzig (1957) showed that an upper bound on KP is

$$u = \lfloor \overline{p}_{b-1} + \frac{r p_b}{w_b} \rfloor, \tag{7}$$

where $\lfloor x \rfloor$ is the greatest integer less than or equal to x. This bound is known as the *Dantzig (upper) bound.*

In the following section we will illustrate some main properties of typical solutions to the Knapsack Problem, and use these results for sketching a new algorithm in section 3. Parts of the algorithm are described in sections 4 to 6, while the main algorithm is sketched in section 7. Computational experiments are presented in section 8, followed by a conclusion.

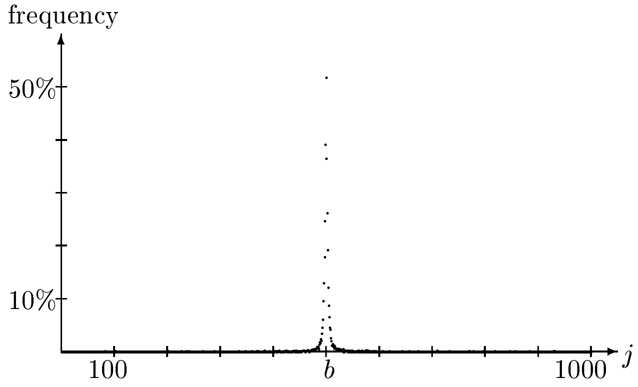

Figure 1: Frequency of items j where the optimal solution x_j differ from the break solution x'_j.

2 Properties of solutions to KP

To illustrate some main properties of solutions to the Knapsack Problem, we have performed the following computational experiment: 1000 data instances were constructed with $n = 1000$, p_j and w_j randomly distributed in $[1, 1000]$, and the capacity c chosen such that $b = 500$ for all instances. Each data instance was completely sorted according to (2), and then solved by a simple branch-and-bound algorithm. We recorded the average number of items where the optimal solution x_j was differing from the break solution x'_j (figure 1) as well as the mean value of the corresponding weights (figure 2).

It turned out, that the number of items where $x_j \neq x'_j$ is only about 3.4 in mean per instance, and as seen from figure 1, such items are generally very close to the break item b. Only a few variables far from b differ from the break solution, and according to figure 2, the weights of the corresponding items decrease as the distance from b grows. From these observations we draw the following conclusions:

- The branching tree of a branch-and-bound algorithm should start with the break solution and gradually enumerate the items from b outwards in a symmetric way.

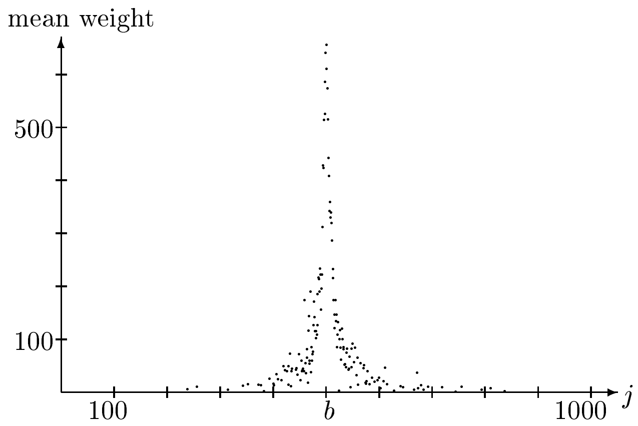

Figure 2: Mean weight of items j where the optimal solution x_j differ from the break solution x'_j.

This will ensure that good solutions will be traversed first.

- It seems that small-weighted items are used for achieving a sufficiently filled knapsack. Thus when choosing branching variables we should strive towards a filled knapsack.

- The solution vector x should be represented as a list of variables differing from the break solution, since this will save updating and space.

3 Basic structure of the algorithm

The p.t. most efficient algorithms for KP are those by Fayard and Plateau (1982), and Martello and Toth (1988). These algorithms are based on the fact that for large problem sizes ($n > 1000$) it turns out that up to 90% of the computing time is spent for the sorting (2). The mentioned algorithms avoid this problem by determining a small subset of the items around the break item b called the *core*, by a modification of the Balas and Zemel (1980) method. The corresponding *core problem* is solved exactly, hoping that all remaining variables x_j may be fixed at their optimal value by some later reduction algorithm. However if some items outside the core cannot be fixed, a new problem containing all remaining free items, must be solved. Such algorithms suffer from the following weaknesses:

- The size of the core is not known before the algorithm has terminated, and therefore is impossible to predict. Several techniques for guessing the core size have been applied, but none are exact.

- The reduction phase is performed at a fixed moment in the algorithm. Since the lower bound is gradually improved by the branch-and-bound algorithm, better reductions can be achieved at a later stage of the solution process. Thus the reduction algorithm should be a dynamic part of the branch-and-bound algorithm.

- The enumeration performed by solving the core problem is only used to achieve a good initial solution, so most of the enumeration has to be repeated, if optimality of the core solution cannot be proved.

These problems can be avoided by letting the core be an interval $[\sigma, \tau]$ around b which is expanded by need. This idea leads to the following algorithm:

a) Find the break item b through a modification of the QUICKSORT algorithm: At each partitioning, the interval not containing the break item b, is added to a list H or L of partitioned intervals.

b) Find a heuristic solution by using a simple greedy algorithm.

c) Set the core to the interval $[b, b]$.

d) Use a branch-and-bound algorithm with a symmetric branching tree for enumerating items in the order $b, b-1, b+1, b-2, \ldots$. Each time the branch-and-bound algorithm grows outside the core, the nearest interval is taken from the list H or L. The interval is reduced by using logical tests to fix some variables at their optimal value, and then partitioned by the same algorithm as in step a) till some more items have been sorted. The core is expanded with these items, and the branching continues.

In section 4 we will present the partitioning algorithm used to find the break item. Section 5 will describe the branch-and-bound algorithm and section 6 the core-expanding algorithm. Finally section 7 describes the chosen heuristic solution, and sketches the main algorithm.

3.1 Roundoff errors

As noted in Pisinger and Walukiewicz (1989), floating-point arithmetics in KP may lead to roundoff errors, since a computer only works with approximations to the real numbers. Especially when using truncating functions (like $\lfloor x \rfloor$ in the Dantzig bound) it may yield dramatic errors even for small deviations, and thus lead to incorrect results.

The problem may be avoided by rewriting expressions in the following way: Testing whether the Dantzig bound is better than some lower bound z may be written:

$$\begin{aligned} \lfloor \overline{p}_{b-1} + \tfrac{rp_b}{w_b} \rfloor &> z && (8)\\ \Leftrightarrow\ \overline{p}_{b-1} + \frac{rp_b}{w_b} &\geq z+1 && (9)\\ \Leftrightarrow\ \det(z+1-\overline{p}_{b-1}, r, p_b, w_b) &\leq 0 && (10) \end{aligned}$$

where $\det(a_1, a_2, b_1, b_2) = a_1 b_2 - a_2 b_1$. Throughout this paper we will rewrite expressions as above to avoid floating-point arithmetics.

4 Finding the break item

It should be clear from the definitions in section 1 that the break item b plays an important role in the solution of KP: Knowing the break item, we may determine the Dantzig upper bound, and we can construct the break solution, which is a quite good lower bound. Moreover knowing these upper and lower bounds, we may reduce the problem size by fixing some variables x_j at their optimal value. Finally the break solution is a good starting point for a branching tree.

The definition of the break item requires a complete sorting of the items according to (2), but this sorting takes up most of the computing time for large problem sizes. However Balas and Zemel (1980) proposed a technique which allows us to determine the break item in $O(n)$ time. This technique is based on a definition of b which only demands a partial sorting of the items, since we define the break item by the following three properties:

$$\overline{w}_{b-1} \leq c < \overline{w}_b, \tag{11}$$

$$e_j \geq e_b, \quad j = 1, \ldots, b-1, \tag{12}$$
$$e_j \leq e_b, \quad j = b, \ldots, n. \tag{13}$$

It is quite easy to modify a sorting algorithm like QUICKSORT (Hoare 1962) for only making the above partial sorting of the items.

The QUICKSORT algorithm repeatedly chooses some middle value λ from the interval $I = [f, l]$, and partition the interval in two parts $[f, i-1]$ and $[i, l]$, so that

$$e_j \geq \lambda, \quad j \in [f, i-1], \tag{14}$$
$$e_j \leq \lambda, \quad j \in [i, l]. \tag{15}$$

If it turns out that $\overline{w}_{i-1} > c$ then b must be in the interval $[f, i-1]$ according to (11), while if $\overline{w}_{i-1} \leq c$ then b must be in the interval $[i, l]$. If at any stage, we denote $\overline{w}_{i-1}$ by W, and $\overline{w}_{f-1}$ by V, we get the following algorithm:

```
procedure partsort(f,l,V);                              (* [f,l] interval, V = w̄_{f-1} *)
if (l − f > 0) then
   m := ⌊f + d/2⌋;
   Swap items f,m,l so that e_f ≥ e_m ≥ e_l.
fi;
if (l − f < 3) then
   φ := f;  ψ := l;  ν := V;                            (* return found interval *)
else
   p̃ := p_m;  w̃ := w_m;                                 (* λ = p̃/w̃ is median of e_f, e_m, e_l *)
   i := f;  j := l;  W := V;                            (* partition [f,l] in two intervals *)
   repeat
      repeat W := W + w_i;  i := i + 1; until (det(p_i, w_i, p̃, w̃) ≤ 0);
      repeat                j := j − 1; until (det(p_j, w_j, p̃, w̃) ≥ 0);
      if (i < j) then swap(i,j); fi;
   until (i > j);                                       (* e_k ≥ λ, k ∈ [f,j] ∧ e_k ≤ λ, k ∈ [i,l] *)
   if (W > c) then L := L ∪ {[i,l]};  partsort(f, i − 1, V);
              else H := H ∪ {[f, i − 1]};  partsort(i, l, W); fi;
fi;
```

The procedure SWAP(i, j) swaps the items corresponding to the indices i and j.

As seen, the algorithm is a usual median-of-three 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, and clearly the break item will be one of these. The found interval is denoted by $[\phi, \psi]$, and the weight sum up to ϕ is $\nu = \overline{w}_{\phi-1}$.

Note that all discarded intervals are added to the lists $H = \{H_1, \ldots, H_h\}$ (items of *higher* efficiency) and $L = \{L_1, \ldots, L_l\}$ (items of *lower* efficiency). When the PARTSORT algorithm terminates these intervals looks like in figure 3. Since each interval is the result

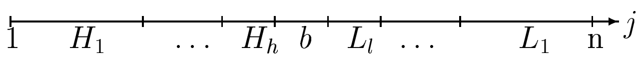

Figure 3: The lists H and L.

of a partitioning by PARTSORT, the efficiency e_j of any item in one interval I will always be greater than or equal to the efficiency of any item in the interval to the right of I. This knowledge will save some computational effort when we have to expand the core.

5 The branch-and-bound algorithm

We will now present a branch-and-bound algorithm EXPBRANCH, which satisfies the requirements stated in section 2. EXPBRANCH is a recursive procedure, which at each recursion will insert (remove) one item to (from) the knapsack. At each recursion, s and t, $s < b \leq t$, indicates that the variables x_j, $j \in [s+1, t-1]$ are fixed to some value, while the remaining variables may be varied arbitrarily. Initially we set $[s,t] = [b-1, b]$.

At any stage P and W determine the profit and weight sum corresponding to the current value of the solution vector $x = \{x_1, \ldots, x_n\}$. Thus: $P = \sum_{j=1}^{n} p_j x_j$, and $W = \sum_{j=1}^{n} w_j x_j$. Our initial vector is the break solution, and each succeeding iteration will try to make W as close to c as possible. Thus if $W \leq c$ we must insert some item $j \geq t$, and if $W > c$ we must remove some item $j \leq s$ from the knapsack. Having inserted (resp. removed) some item j, we call EXPBRANCH recursively, but now with $t = j+1$ (resp. $s = j-1$), meaning that from now on we may only insert items after j (resp. remove items before j). If s or t grows outside of the core $[\sigma, \tau]$, we sort some more items, and expand $[\sigma, \tau]$ correspondingly.

We backtrack if the upper bound does not exceed z, i.e. if

$$\lfloor P + \frac{(c-W)p_t}{w_t} \rfloor \leq z, \quad \text{when } W \leq c, \tag{16}$$

$$\lfloor P + \frac{(c-W)p_s}{w_s} \rfloor \leq z, \quad \text{when } W > c, \tag{17}$$

where the bounds on the left side are found by linear relaxation of x_s and x_t. The tests are equivalent to

$$\det(P-z-1, W-c, p_t, w_t) < 0, \quad \text{when } W \leq c, \tag{18}$$

$$\det(P-z-1, W-c, p_s, w_s) < 0, \quad \text{when } W > c. \tag{19}$$

If no bound stops the branching, s may grow below 1, or t may grow beyond n. In such case we must backtrack, since no more items can be inserted or removed. This is ensured by expanding the data instance with two stop items 0 and $n+1$ where $(p_0, w_0) = (1, 0)$ and $(p_{n+1}, w_{n+1}) = (0, 1)$. These limits ensure that when reaching $s = 0$ or $t = n+1$ the upper bounds will be so bad, that we are forced to backtrack.

Instead of keeping track of the current vector x, we only keep track of the items j differing from the break solution x', i.e. items where $x_j \neq x'_j$. Such items are added to an *exception list* E each time some improved solution has been found. Since the number of such exceptions according to section 2 usually is very small, this technique saves a great amount of updating. When the algorithm terminates, we first set x to the break solution x', and then we change the x_j-values according to the exception list.

Now it should be straightforward to understand the following sketch of EXPBRANCH:

```
function expbranch(P, W, s, t): boolean;
var improved: boolean;
improved := false;
if (W ≤ c) then (* insert some item j ≥ t *)
  if (P > z) then (* better solution found *) improved := true;  z := P;  E := ∅; fi;
  repeat
    if (t > τ) then expand(L_l);  L := L \ {L_l};  fi;
    if (det(P − z − 1, W − c, p_t, w_t) < 0) then return(improved); fi; (* u.b. test *)
    if expbranch(P + p_t, W + w_t, s, t + 1) then improved := true;  E := E ∪ {t};  fi;
    t := t + 1;
  forever;
else (* remove some item j ≤ s *)
  repeat
    if (s < σ) then expand(H_h);  H := H \ {H_h};  fi;
    if (det(P − z − 1, W − c, p_s, w_s) < 0) then return(improved); fi; (* u.b. test *)
    if expbranch(P − p_s, W − w_s, s − 1, t) then improved := true;  E := E ∪ {s};  fi;
    s := s − 1;
  forever;
fi;
```

Not all feasible solutions are generated by the branching tree, since we only consider solutions close to a filled knapsack:

Proposition 1 *The branching tree of* EXPBRANCH *will reach an optimal solution.*

Proof Assume, that an optimal solution is given by x^*. Then we can write this solution as exceptions from the break solution, yielding a set E^* of the corresponding items. We want to show that this set will be generated by EXPBRANCH.

Starting from the break solution, we insert or remove items from the knapsack according to the following rule: If the weight sum W is less than c we insert the item $j \in E^*$ with lowest index greater than or equal to b, and set $E^* = E^* \setminus \{j\}$. In the same way we remove the item $j \in E^*$ with largest index less than b, if W is greater than c, and set $E^* = E^* \setminus \{j\}$.

Since the solution x^* is optimal, no bounding will stop this iteration. So the process will only stop if one of the following three situations occurs:

$$E^* = \emptyset, \tag{20}$$

$$W > c \quad \wedge \quad \forall j \in E^* \neq \emptyset : j \geq b, \tag{21}$$

$$W \leq c \quad \wedge \quad \forall j \in E^* \neq \emptyset : j < b. \tag{22}$$

Here (20) means that we have reached the optimal solution, while (21) means that x^* is not a solution, and (22) means that x^* is not an optimal solution, since it may be improved by setting $x_j = 1$ for remaining $j \in E^*$. □

Example

We consider the following completely sorted problem with stop items added:

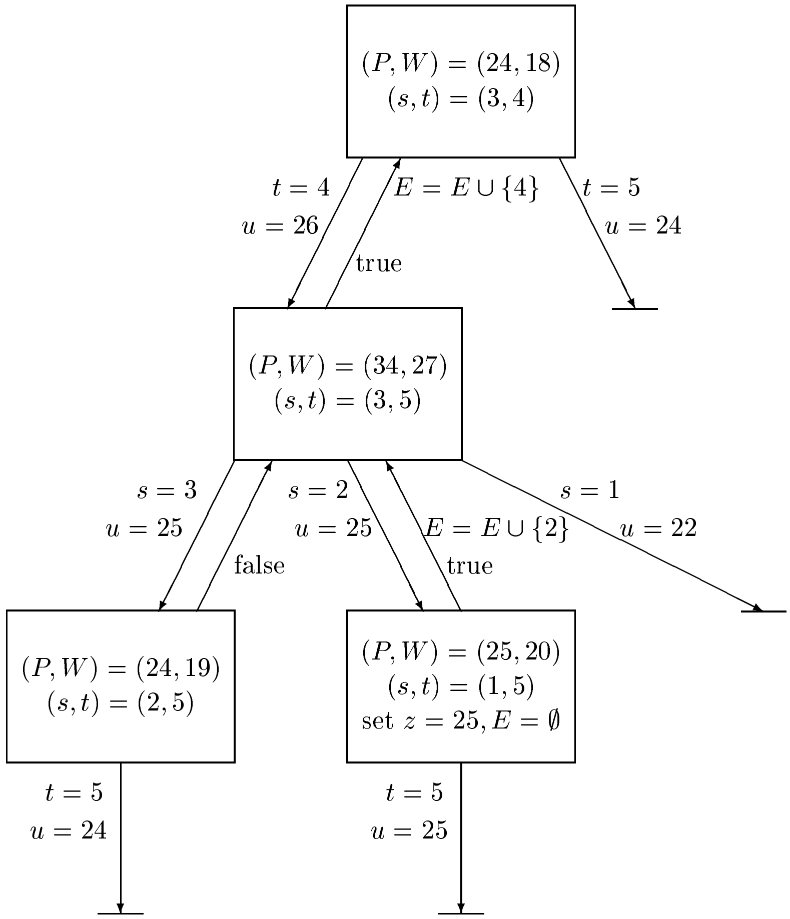

Figure 4: Branching tree for the presented example.

j	0	1	2	3	4	5	6	7
p_j	1	5	9	10	10	2	1	0
w_j	0	3	7	8	9	5	3	1

Where we have $n = 6,\ c = 20,\ b = 4$, and $[\sigma, \tau] = [0, 7]$.

In figure 4, we show the corresponding branching tree, which is traversed top-down in left-right direction. Each box corresponds to one procedure call, while each downgoing arrow determines one iteration within the main loop of the procedure, and each upgoing arrow illustrates the return from a sub-iteration. To improve understandability, upper bounds u are determined by the original formulas (16) and (17). As initial lower bound we chose $z = \overline{p}_{b-1} = 24$.

Upon termination we find the solution vector by first setting $x_j = 1,\ j = 1, \ldots, 3$ and $x_j = 0,\ j = 4, \ldots, 6$, and then swapping the x_j values according to the exception list $E = \{2, 4\}$. The final solution vector is $x = (101100)$, and the corresponding objective value is $z = 25$.

6 Expanding the core by need

Each time the EXPBRANCH algorithm reaches the border of the core, we have to expand the core. This is done by choosing an interval from the list H (resp. L) which is closest to the break item, and using the PARTSORT algorithm to partition the interval. But before doing this, we try to reduce the interval size by checking whether some variables x_j can be fixed at their optimal value.

6.1 Problem reduction by preprocessing

Ingargiola and Korsh (1973) presented the first reduction algorithm for the Knapsack Problem, but many improved versions have been developed since then: For instance Dembo and Hammer (1980), Fayard and Plateau (1982), and Martello and Toth (1988).

All these reduction algorithms are based on the fact, that if an upper bound on KP with additional constraint $x_j = \alpha, \alpha \in \{0, 1\}$ is less than or equal to some lower bound z, we may conclude that the branch $x_j = \alpha$ never will lead to an improved solution, and thus can fix x_j at $1 - \alpha$.

More formally let u_j^0 (resp. u_j^1) be any upper bound on KP with the additional constraint $x_j = 0$ (resp. $x_j = 1$), and let z be a lower bound on KP. Then the general reduction scheme is:

$$\begin{aligned} u_j^0 \le z \quad &\Rightarrow \quad x_j = 1, \\ u_j^1 \le z \quad &\Rightarrow \quad x_j = 0. \end{aligned} \tag{23}$$

The strength of such a reduction clearly depends on how tight the upper and lower bounds are. As a lower bound we can use the best solution found so far by the branch-and-bound algorithm, or initially just use some heuristic solution. As an upper bound we choose the bound by Dembo and Hammer (1980):

$$\begin{aligned} u_j^0 &= \lfloor \overline{p}_{b-1} - p_j + \frac{(r + w_j)p_b}{w_b} \rfloor, \ j = 1, \dots, b-1, \\ u_j^1 &= \lfloor \overline{p}_{b-1} + p_j + \frac{(r - w_j)p_b}{w_b} \rfloor, \ j = b, \dots, n, \end{aligned} \tag{24}$$

where $r = c - \overline{w}_{b-1}$ is the residual capacity. These bounds are not the most tight, but they only demand that the break item b is defined by (11), and that the items are ordered according to (12) and (13). Since the reduction is performed dynamically throughout the solution process, we may expect that a better solution and thus a better lower bound is found during the branching. This will usually more than compensate for the quite weak upper bounds.

Using the bounds by Dembo and Hammer, we may rewrite the tests (23) as:

$$\begin{aligned} \det(z + 1 - \overline{p}_{b-1} + p_j, r + w_j, p_b, w_b) > 0 \quad &\Rightarrow \quad x_j = 1, \quad j = 1, \dots, b-1, \\ \det(z + 1 - \overline{p}_{b-1} - p_j, r - w_j, p_b, w_b) > 0 \quad &\Rightarrow \quad x_j = 0, \quad j = b, \dots, n, \end{aligned} \tag{25}$$

which yields the following sketch of the reduction:

```
procedure reduce(var i, j);                              (* [i,j] interval to be reduced *)
if (j < b) then                                          (* [i,j] is left of [σ,τ] *)
  k := σ − 1;
  while (i ≤ j) do
    if (det(z + 1 − p̄_{b−1} + p_j, r + w_j, p_b, w_b) > 0) then   (* test item j *)
      swap(i, j);  i := i + 1;                           (* x_j fixed to 1 *)
    else
      swap(j, k);  j := j − 1;  k := k − 1;              (* x_j free *)
    fi;
  endwhile;
  if (k = σ − 1) then swap(j, k);  k := k − 1;  fi;      (* return at least one item *)
  i := k + 1;  j := σ − 1;
else                                                     (* [i,j] is right of [σ,τ] *)
  k := τ + 1;
  while (i ≤ j) do
    if (det(z + 1 − p̄_{b−1} − p_i, r − w_i, p_b, w_b) > 0) then   (* test item i *)
      swap(i, j);  j := j − 1;                           (* x_i fixed to 0 *)
    else
      swap(i, k);  i := i + 1;  k := k + 1;              (* x_i free *)
    fi;
  endwhile;
  if (k = τ + 1) then swap(i, k);  k := k + 1; fi;       (* return at least one item *)
  i := τ + 1;  j := k − 1;
fi;
```

The algorithm receives an interval $[i, j]$ as argument, and terminates with $[i, j]$ set to the reduced interval of free items. The returned interval will always contain at least one item even if the corresponding variable x_j could be fixed by the criteria (25). This ensures that the branch-and-bound algorithm always gets an item for deriving upper bounds, which usually will save some computational effort: If the branch-and-bound algorithm backtracks, no further sorting or reduction is needed. Note how the items are swapped to ensure that the interval $[i, j]$ of free items always is placed immediately beside the core $[\sigma, \tau]$.

The two while-loops are performed in linear time, which means that each reduction of an interval $[i, j]$ is performed in $O(j - i + 1)$. Since the reduction algorithm is applied each time we expand the core, the computational complexity of all reductions corresponds to the complexity of a complete sorting. And QUICKSORT has an average computational complexity $O(n \log n)$.

6.2 Sorting

Having reduced the interval size, we must sort the corresponding items according to (2). For this purpose we use the PARTSORT algorithm, since it repeatedly will partition the intervals closest to b until at most three items have been sorted. The unsorted intervals will be added to the lists H and L, and can be used next time the core needs to be expanded.

This lead us to the following sketch of the core-expanding algorithm:

```
procedure expand(i, j);                                  (* [i, j] interval from H or L *)
reduce(i, j);                                            (* do the reduction *)
if (j < b) then partsort(i, j, 0) else partsort(i, j, c); fi;   (* choose direction for sorting *)
σ := min(σ, φ);  τ := max(τ, ψ);                         (* adjust core limits *)
```

7 Heuristic solution and main algorithm

Expknap needs some kind of heuristic solution to act as initial lower bound for the branch-and-bound algorithm. Purists may use $z = \overline{p}_{b-1}$, but quite a lot of unnecessary branching may be saved by choosing a better heuristic solution.

Since the items only are ordered according to (12) and (13), we use the following heuristic solutions: The *forward greedy solution*, which is the best value of the objective function when adding one item to the break solution:

$$z_f \quad = \quad \max_{j=b,\dots,n} \{\overline{p}_{b-1} + p_j \mid \overline{w}_{b-1} + w_j \le c\}, \tag{26}$$

and the *backward greedy solution*, which is the best value of the objective function when adding the break item to the break solution and removing another item:

$$z_b \quad = \quad \max_{j=1,\dots,b-1} \{\overline{p}_b - p_j \mid \overline{w}_b - w_j \le c\}. \tag{27}$$

We choose the best of z_f and z_b as our lower bound z, and store the corresponding solution vector as exceptions E from the break solution.

The main algorithm may now be sketched in the following way:

```
function expknap(p, w, x, c, n): integer;
p_0 := 1;  w_0 := 0;  H := {[0, 0]};                     (* add stop items *)
p_{n+1} := 0;  w_{n+1} := 1;  L := {[n + 1, n + 1]};
partsort(1, n, 0);  [σ, τ] := [φ, ψ];                     (* find initial core *)
b := φ;  r := c − ν;                                      (* determine b *)
while (w_b ≤ r) do r := r − w_b;  b := b + 1; endwhile;
z := max(z_f, z_b); Set E to the corresponding vector.    (* determine lower bound *)
expbranch(p̄_{b−1}, w̄_{b−1}, b − 1, b);                   (* branch-and-bound *)
Define optimal solution x from E.                         (* solution is defined *)
return(z);
```

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 how the algorithm behaves for different problem sizes, test instances, and data-ranges. Four types of randomly generated data instances are considered as listed below. Each type will be tested with *data-range* R =100, 1000, 10000 for different problem sizes n =50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000. The capacity c is chosen as $c = \frac{1}{2}\overline{w}_n$.

n	50	100	200	500	1000	2000	5000	10000	20000	50000	100000
PARTSORT	0	0	0	0	1	1	4	9	17	57	128
SORT	0	1	1	2	4	8	26	55	121	361	834

Table 1: Sorting times in milliseconds. Average of 50 instances.

- *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. If some data instance was not solved within 1 hour the field is marked with a "—".

To get a deeper understanding of the algorithm, we measure the efficiency of each part of the algorithm: Finding the break solution, reducing the data instance, branching and bounding. Moreover we show how large the core grew, and give the average total computing time for EXPKNAP. Finally the computing times are compared to those of the MT2 algorithm which has been obtained from Martello and Toth (1990).

First table 1 shows how much time is spent for finding the break solution by using the PARTSORT algorithm compared to a complete sorting SORT. Since the sorting does not depend on the problem type, we only show the average times for uncorrelated data instances. As expected the computing time of PARTSORT grows linearly with the problem size, and for large problem sizes PARTSORT is considerably faster than a complete sorting.

Next table 2 shows how often the reduction criteria (25) has been applied as a percentage of all the items. Since the reduction criteria may be applied several times for each item, this percentage may grow beyond 100 percent. It is seen, that small-sized data instances generally apply the reduction criteria more often than large-sized data instances,

	uc			wc			sc			ss		
n	100 [a]	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
50	81	90	88	107	123	122	130	198	221	78	190	202
100	82	87	93	83	140	128	142	234	269	64	222	255
200	69	89	99	53	125	128	—	—	—	21	232	300
500	51	84	95	19	123	121	—	—	—	1	176	338
1000	29	88	97	20	112	120	—	—	—	0	87	375
2000	11	91	100	9	100	113	—	—	—	0	24	349
5000	1	91	97	1	30	110	—	—	—	0	0	195
10000	0	74	101	0	10	110	—	—	—	0	0	120
20000	0	56	102	0	6	106	—	—	—	0	0	9
50000	0	13	97	0	3	92	—	—	—	0	0	0
100000	0	4	96	0	3	12	—	—	—	0	0	0

[a] These values indicate the data-range $R = 100, 1000$ or 10000.

Table 2: Use of reduction criteria as percentage of n. Average of 50 instances.

	uc			wc			sc			ss		
n	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
50	0	1	1	2	7	4	1234	1423	8888	1	8	122
100	1	1	2	2	20	17	17780	476874	373863	0	8	115
200	1	2	3	1	34	57	—	—	—	0	7	118
500	1	6	14	1	102	188	—	—	—	0	4	87
1000	1	21	34	1	136	541	—	—	—	0	3	99
2000	0	58	94	0	135	1167	—	—	—	0	1	59
5000	0	110	336	0	56	4031	—	—	—	0	0	49
10000	0	107	1048	0	99	7071	—	—	—	0	0	23
20000	0	70	2026	0	146	6658	—	—	—	0	0	2
50000	0	15	4446	0	71	5853	—	—	—	0	0	0
100000	0	12	7234	0	3	3610	—	—	—	0	0	0

Table 3: Number of iterations performed by EXPBRANCH in hundreds. Average of 50 instances.

and that the use grows with the correlation and data-range. This may be explained by the fact, that a solution close to the Dantzig bound is found easily for large-sized and low-correlated data instances.

Table 3 shows the number of recursive procedure calls performed by EXPBRANCH. It is seen, that large-ranged data instances generally use most iterations, and that the number of iterations grows with the correlation (although subset-sum problems are quite easy).

Table 4 shows the final core size as a percentage of the problem size. The core size is generally very small but seems to grow with the data-range. This explains why it is almost impossible to give an a priori size of the core, like Martello and Toth (1988) tried.

The average computing time for each data instance is given in table 5. It is seen, that EXPBRANCH is able to solve most data instances with up to 100000 items, within one second. Only the strongly correlated data instances cause problems, but this could be expected, since strongly correlated data instances are really hard (Balas and Zemel (1980)).

Finally table 6 gives the corresponding computing times using the MT2 algorithm. The same instances have been tested on the same computer for both algorithms. It is seen, that MT2 does not behave in a regular way: Some large sized instances were not solved within hours (or even days) while the remaining instances were solved in seconds. This behavior is a consequency of the a priori detection of the core: If a good lower

	uc			wc			sc			ss		
n	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
50	21.8	25.8	24.0	39.3	48.9	46.5	56.8	91.4	99.4	40.1	92.7	98.8
100	14.1	14.8	16.7	21.5	41.9	37.6	49.6	83.5	97.1	25.5	86.6	100.0
200	8.5	9.5	11.2	10.0	25.7	26.6	—	—	—	7.2	74.9	98.0
500	3.9	5.5	7.0	3.9	17.2	16.1	—	—	—	0.5	43.2	85.4
1000	2.1	4.1	4.6	2.7	10.7	12.3	—	—	—	0.2	18.6	83.9
2000	0.8	3.4	3.2	1.0	7.3	7.7	—	—	—	0.1	4.0	69.7
5000	0.2	2.0	2.0	0.2	2.1	5.4	—	—	—	0.1	0.1	34.1
10000	0.1	1.2	1.6	0.0	0.9	3.8	—	—	—	0.0	0.0	18.8
20000	0.0	0.8	1.0	0.0	0.5	2.5	—	—	—	0.0	0.0	1.2
50000	0.0	0.3	0.6	0.0	0.3	1.8	—	—	—	0.0	0.0	0.0
100000	0.0	0.1	0.5	0.0	0.1	0.6	—	—	—	0.0	0.0	0.0

Table 4: Final core size as percentage of n. Average of 50 instances.

n	uc			wc			sc			ss		
	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
50	0	0	0	0	2	0	172	194	1249	0	2	17
100	0	0	0	0	2	3	35672	66171	30738	0	2	16
200	0	1	1	1	5	9	—	—	—	1	2	17
500	1	2	4	1	15	27	—	—	—	0	3	15
1000	1	4	7	1	20	77	—	—	—	1	3	20
2000	2	11	16	2	22	165	—	—	—	2	3	20
5000	5	23	54	5	14	566	—	—	—	6	4	28
10000	9	28	160	9	26	994	—	—	—	9	8	32
20000	20	36	313	20	44	958	—	—	—	18	16	20
50000	67	85	724	64	87	928	—	—	—	50	50	52
100000	163	194	1278	161	185	712	—	—	—	110	111	112

Table 5: Total computing times in milliseconds (EXPKNAP). Average of 50 instances.

n	uc			wc			sc			ss		
	100	1000	10000	100	1000	10000	100	1000	10000	100	1000	10000
50	0	0	0	0	0	0	37	16	76	0	0	8
100	0	1	0	0	1	0	10590	5210	21145	0	1	12
200	0	0	2	0	3	2	—	—	—	0	2	19
500	2	2	3	1	7	11	—	—	—	0	2	20
1000	2	4	6	2	8	20	—	—	—	1	3	22
2000	4	9	13	13	10	41	—	—	—	0	2	22
5000	8	21	35	6	16	87	—	—	—	4	6	24
10000	17	28	65	14	46	155	—	—	—	9	11	26
20000	40	46	122	31	81	229	—	—	—	20	22	38
50000	—	139	329	113	301	310	—	—	—	59	60	81
100000	—	282	577	254	—	1326	—	—	—	122	129	146

Table 6: Total computing times in milliseconds (MT2). Average of 50 instances.

bound cannot be determined by solving the core problem, an overwhelming branching tree attends.

The two algorithms are generally equally fast, but EXPKNAP has a very stable behavior, with no anomalous instances. Thus we may conclude, that EXPKNAP dominates MT2. For a comparison of MT2 to other algorithms see Martello and Toth (1990).

9 Conclusions

We have presented a complete algorithm for the exact solution of the 0-1 Knapsack Problem.

The presented results show that EXPKNAP is one of the most efficient algorithms available in the literature. The symmetric branching tree and the lazy sorting and reduction imply that most data instances are solved without sorting or even reducing most of the items. And the expanding core avoids the anomalous behaviour of the "fixed-core" algorithms. Moreover we have demonstrated that all calculations can be performed with integers, thus avoiding the risk of deriving wrong solutions.

Since the algorithm is so simple to implement (only about 200 lines) it should be an attractive alternative to algorithms like MT2 which has about 1400 lines. Small-sized problem instances may be solved even easier, by sorting all items and only using the EXPBRANCH algorithm. This algorithm would be extremely simple, and still yield a good performance.

References

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

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.

Dudzinski, K. and S. Walukiewicz, "Exact Methods for the Knapsack Problem and its Generalizations," *European Journal of Operational Research*, 28 (1987), 3-21.

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.

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, "Algorithms for Knapsack Problems," in S. Martello, G. Laporte, M. Minoux and C. Ribeiro (Eds.), *Surveys in Combinatorial Optimization*, Ann. Discrete Math. 31, North-Holland, Amsterdam, 1987, 213-257.

Martello, S. and P. Toth, "A New Algorithm for the 0-1 Knapsack Problem," *Management Science*, 34 (1988), 633-645.

Martello, S. and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations,* John Wiley & Sons Ltd. England, 1990.

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

Pisinger, D. and S. Walukiewicz, "Experiments with 0-1 Knapsack Problem Algorithms," Working Paper ZPM 32/89, Systems Research Institute, Warsaw (1989).