Journal of Algorithms **33,** 1–14 (1999)
Article ID jagm.1999.1034, available online at http://www.idealibrary.com on IDEAL®

Linear Time Algorithms for Knapsack Problems with Bounded Weights

David Pisinger*

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

Received August 21, 1996; revised October 8, 1998

A new technique called *balancing* is presented for the solution of Knapsack Problems. It is proved that an optimal solution to the Knapsack Problem is balanced, and thus only balanced feasible solutions need to be enumerated in order to solve the problem to optimality. Restricting a dynamic programming algorithm to only consider balanced states implies that the Subset-sum Problem, 0–1 Knapsack Problem, Multiple-choice Subset-sum Problem, and Bounded Knapsack Problem all are solvable in linear time, provided that the weights and profits are bounded by a constant. Extensive computational experiments are presented to document that the derived algorithm for the Subset-sum Problem is able to solve several problems from the literature which could not be solved previously.

Key Words: Knapsack Problem; dynamic programming; memorizing.

1. INTRODUCTION

We will consider different variants of the 0–1 Knapsack Problem (KP) given by

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \le C, \\ & x_j \in \{0,1\}; \qquad j = 1,\ldots,n, \end{aligned} \tag{1}$$

* E-mail: pisinger@diku.dk

0196-6774/99 $30.00

where p_j and w_j are positive integers representing the profits and weights of some items, while C is the capacity of the knapsack. To avoid trivial cases we assume $\sum_{j=1}^{n} w_j > C$ and $w_j \leq C$ for $j = 1, \ldots, n$.

Branch-and-bound algorithms for Knapsack Problems have attained much interest during the last two decades due to their ability to solve several practically occurring problems. It is, however, not possible to give reasonable time bounds for these algorithms, as instances may be constructed which demand complete enumeration. If better time bounds are demanded, two techniques appear in the literature: *Dynamic programming* (Bellman [2]) makes it possible to solve a KP in *pseudo polynomial* time $O(nC)$, which is attractive for moderate capacities. Horowicz and Sahni [8] introduced the *partitioning* technique, which is useful if the capacity is very large. The items are partitioned into two sets, each of which is enumerated completely, and then the obtained states are paired in linear time, leading to a complexity of $O(2^{n/2})$.

In this paper we present a third technique *balancing* which implies that the enumeration may be restricted to those feasible solutions where the corresponding weight sum in some respect is sufficiently close to the capacity of the knapsack. When a dynamic programming algorithm only needs to consider balanced solutions, tighter time bounds than previously may be derived. Among the main results should be emphasized that—provided all profits and weights are bounded by a constant—the Subset-sum Problem, 0−1 Knapsack Problem, Multiple-choice Subset-sum Problem, and Bounded Knapsack Problem can all be solved in linear time. The assumption of bounded coefficients is realistic for many instances, as e.g., most computational experiments in the literature consider such problems.

In Section 2 we prove that an optimal solution to KP is a balanced filling, and thus, any enumerative algorithm may be restricted to consider balanced solutions. In the following three sections we present balanced algorithms for the Subset-sum Problem, 0−1 Knapsack Problem, and other variants of the problem, discussing further applications at the end of Section 5. Finally Section 6 experimentally compares different algorithms for solving the Subset-sum Problem, showing that the presented technique makes it possible to solve several problems that could not be handled previously.

2. BALANCED OPERATIONS

In the following definitions we do not assume any particular ordering of the items. Let the *break item* b be the first item which does not fit into the knapsack when including the items successively, thus $b = \min\{j : \sum_{i=1}^{j} w_i >$

$C\}$. The *break solution* x' is the feasible solution obtained by including items up to b in the knapsack; thus, $x'_j = 1$, $j = 1, \ldots, b-1$, and $x'_j = 0$, $j = b, \ldots, n$. The weight sum corresponding to the break solution is $\overline{w} = \sum_{j=1}^{b-1} w_j$.

DEFINITION 1. A *balanced filling* is a solution x obtained from the break solution through *balanced operations* as

- The break solution x' is a balanced filling.
- *Balanced insert*. If we have a balanced filling x with $\sum_{j=1}^{n} w_j x_j \leq C$ and change a variable x_t $(t \geq b)$ from $x_t = 0$ to $x_t = 1$ then the new filling is also balanced.
- *Balanced remove*. If we have a balanced filling x with $\sum_{j=1}^{n} w_j x_j > C$ and change a variable x_s $(s < b)$ from $x_s = 1$ to $x_s = 0$ then the new filling is balanced.

PROPOSITION 1. *An optimal solution to KP is a balanced filling*; *i.e., it may be obtained through balanced operations*.

Proof. Assume that the optimal solution is given by x^*. Let $s_1, \ldots, s_\alpha$ be the indices $s_i < b$, where $x^*_{s_i} = 0$, and $t_1, \ldots, t_\beta$ be the indices $t_i \geq b$, where $x^*_{t_i} = 1$. Order the indices such that $s_\alpha < \cdots < s_1 < b \leq t_1 < \cdots < t_\beta$.

Starting from the break solution $x = x'$ we perform balanced operations in order to reach x^*. As the break solution satisfies that $\sum_{j=1}^{n} w_j x_j \leq C$ we must insert item t_1, thus set $x_{t_1} = 1$. If the hereby obtained weight sum $\sum_{j=1}^{n} w_j x_j$ is greater than C we remove item s_1 by setting $x_{s_1} = 0$, otherwise we insert the next item t_2. Continue this way till one of the following three situations occur:

(1) All the changes corresponding to $\{s_1, \ldots, s_\alpha\}$ and $\{t_1, \ldots, t_\beta\}$ were done, meaning that we reached the optimal solution x^* through balanced operations.

(2) We reach a situation where $\sum_{j=1}^{n} w_j x_j > C$ and all indices $\{s_i\}$ have been used but some $\{t_i\}$ have not been used. This implies that x^* is not a feasible solution since the current solution is infeasible and we still need to insert some items to obtain x^*.

(3) A similar situation where $\sum_{j=1}^{n} w_j x_j \leq C$ is reached and all indices $\{t_i\}$ have been used, but some $\{s_i\}$ are missing. This implies that x^* cannot be an optimal solution, since the current solution x is feasible and has a better profit sum. ∎

COROLLARY 1. *An optimal solution may be obtained through balanced operations by considering the indices* $\{t_i\}$ *in increasing order, and the indices* $\{s_i\}$ *in decreasing order*.

COROLLARY 2. *Any balanced filling x satisfies the following bound on the corresponding weight sum*: $C - W < \sum_{j=1}^{n} w_j x_j \leq C + W$, *where* $W = \max_{j=1,\ldots,n} w_j$.

3. A BALANCED ALGORITHM FOR THE SUBSET-SUM PROBLEM

The *Subset-sum Problem* (*SSP*) is an ordinary KP, where the profits and weights satisfy that $p_j = w_j$, $j = 1, \ldots, n$. Bellman [2] presented a dynamic programming algorithm for SSP which runs in $O(nC)$. If all weights w_j are bounded by a fixed constant W, this complexity may be written $O(n^2W)$, i.e. quadratic time for constant W. We will present an algorithm running in $O(nW)$ based on balancing. Due to the weight constraint in Corollary 2 let $f_{s,t}(\tilde{c})$ $(s \leq b,\ t \geq b - 1,\ c - W < \tilde{c} \leq c + W)$ be an optimal solution to the subproblem of SSP, which is defined on the variables $i = s, \ldots, t$ of the problem:

$$f_{s,t}(\tilde{c}) = \max\left\{ \begin{array}{l} \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j : \\ \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j \leq \tilde{c}, \\ x_j \in \{0,1\} \text{ for } j = s, \ldots, t, \\ x \text{ is a balanced filling} \end{array} \right\}, \tag{2}$$

where we define $\sum_{j=s}^{t} w_j = 0$ if $s > t$. We will only consider those *states* (s, t, μ), where $\mu = f_{s,t}(\mu)$, i.e., those weight sums μ which can be obtained by balanced operations on $x_s, \ldots, x_t$, applying the following (unusual) dominance relation:

DEFINITION 2. Given two states (s, t, μ) and (s', t', μ'). If $\mu = \mu'$, $s' \leq s$ and $t \leq t'$, then state (s, t, μ) *dominates* state (s', t', μ').

If a state (s, t, μ) dominates another state (s', t', μ') then we may discard the latter. Using the dominance rule, we will enumerate the states for t running from $b - 1$ to n. Thus, at each stage t and for each value of μ we will have only one index s, which actually is the largest s, such that a balanced filling with weight sum μ can be obtained at the variables $x_s, \ldots, x_t$. Therefore, let $s_t(\mu)$ for $t = b - 1, \ldots, n$ and

$C - W < \mu \le C + W$ be defined as

$$s_t(\mu) = \max s \begin{cases} \text{there exists a balanced filling } x \text{ which satisfies} \\ \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j = \mu;\ x_j \in \{0,1\}, \quad j = s, \ldots, t, \end{cases} \tag{3}$$

where we set $s_t(\mu) = 0$ if no balanced filling exists. At the initial stage, where $t = b - 1$, only one value of $s_t(\mu)$ is positive, namely $s_t(\overline{w}) = b$, as only the break solution is a balanced filling. An optimal solution to SSP is found at the last stage as $z = \max\{\mu \le C : s_n(\mu) > 0\}$.

After each iteration of t we will ensure that all states are feasible by removing sufficiently many items $j < s_t(\mu)$ from those solutions, where $\mu > C$. Thus, only states $s_t(\mu)$ with $\mu \le C$ need to be saved.

To improve efficiency, we will use memorization [5] in connection with the straightforward dynamic programming. Thus, for a given state with $\mu > C$ the table $s_t(\mu)$ indicates that the removal of items $j < s_t(\mu)$ already has been performed such that repeated removals can be avoided. This leads to the following algorithm.

1. ALGORITHM `balsub`.
2. **for** $\mu \leftarrow C - W + 1$ **to** C **do** $s_{b-1}(\mu) \leftarrow 0$;
3. **for** $\mu \leftarrow C + 1$ **to** $C + W$ **do** $s_{b-1}(\mu) \leftarrow 1$;
4. $s_{b-1}(\overline{w}) \leftarrow b$;
5. **for** $t \leftarrow b$ **to** n **do**
6. **for** $\mu \leftarrow C - W + 1$ **to** $C + W$ **do** $s_t(\mu) \leftarrow s_{t-1}(\mu)$;
7. **for** $\mu \leftarrow C - W + 1$ **to** C **do** $\mu' \leftarrow \mu + w_t$; $s_t(\mu') \leftarrow \max\{s_t(\mu'), s_{t-1}(\mu)\}$;
8. **for** $\mu \leftarrow C + w_t$ **downto** $C + 1$ **do**
9. **for** $j \leftarrow s_t(\mu) - 1$ **downto** $s_{t-1}(\mu)$ **do** $\mu' \leftarrow \mu - w_j$; $s_t(\mu') \leftarrow \max\{s_t(\mu'), j\}$;

Algorithm `balsub` does the following (see Fig. 1 for an example): For $t = b - 1$ we only have one balanced solution, the break solution; thus $s_t(\mu)$ is initialized according to this in lines 2–4. Since $s_t(\mu)$ for $\mu > C$ is used for memorizing, states with $\mu > C$ are set to $s_t(\mu) = 1$ as no items have been removed yet.

Now we consider the items $t = b, \ldots, n$ in lines 5–9. In each iteration item t may be added to the knapsack or omitted. Line 6 corresponds to the latter case, thus the states $s_{t-1}(\mu)$ are copied to $s_t(\mu)$ without changes. Line 7 adds item t to each feasible state, obtaining the weight μ'. According to (3), $s_t(\mu')$ is the maximum of the previous value and the current balanced solution.

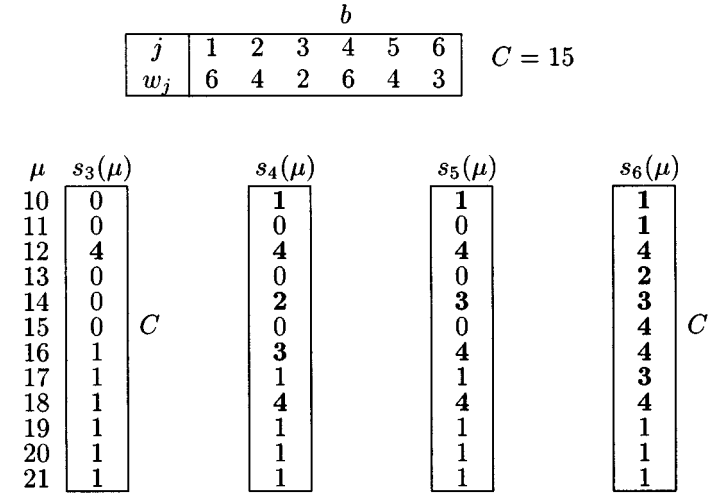

FIG. 1. A given instance and the corresponding table $s_t(\mu)$.

In lines 8–9 we complete the balanced operations by removing items $j < s_t(\mu)$ from states with $\mu > C$. As it may be necessary to remove several items in order to maintain feasibility of the solution, we consider the states for decreasing μ, thus allowing for several removals.

PROPOSITION 2. *Algorithm* `balsub` *finds the optimal solution* x^*.

Proof. We just need to show that the algorithm performs unrestricted balanced operations: (1) It starts from the break solution x'. (2) For each state with $\mu \leq c$ we perform a balanced insert, as each item t may be added or omitted. (3) For each state with $\mu > C$ we perform a balanced remove by removing an item $j < s_t(\mu)$. As the hereby obtained weight μ' satisfies that $\mu' < \mu$ and the weights μ are considered in decreasing order in line 8, we are able to perform multiple removals for each insertion. After each iteration all states are again feasible (i.e., $\mu \leq C$), meaning that a new insertion can be considered.

The only restriction in balanced operations is line 9, where we pass by items $j < s_{t-1}(\mu)$ when items are removed. But due to the memorizing we know that items $j < s_{t-1}(\mu)$ have been removed once before, meaning that $s_t(\mu - w_j) \geq j$ for $j = 1, \ldots, s_{t-1}(\mu)$. Thus repeating the same operations will not contribute to an increase in $s_t(\mu - w_j)$. ∎

PROPOSITION 3. *The complexity of Algorithm* `balsub` *is* $O(nW)$ *in time and space.*

Proof. *Space*: The array $s_t(\mu)$ has size $(n - b + 1)(2W)$, thus $O(nW)$. *Time*: Lines 2–3 demand $2W$ operations. Line 6 is executed $2W(n - b + 1)$ times. Line 7 is executed $W(n - b + 1)$ times. Finally, for each $\mu > C$, line 9 is executed totally $s_n(\mu) \le b$ times. Thus, during the whole process, line 9 is executed at most Wb times. In total, one gets $O(nW)$. ∎

The disadvantage to `balsub` is that we use dynamic programming by *pulling*, meaning that all $2W$ states are considered at each iteration. An algorithm based on dynamic programming by *reaching* (see Ibaraki [9] for definitions) may be obtained by only considering those states where $s_t(\mu) \neq 0$ for $\mu \le C$ and $s_t(\mu) \neq 1$ for $\mu > C$. The states are stored in a data structure [1, 12], supporting the basic operations `search`, `insert`, and `predecessor`. Notice that `predecessor` actually is necessary, as the states with $\mu > c$ have to be considered in decreasing order in line 8.

It is possible to modify the `balsub` algorithm such that only the basic operations `search` and `insert` are necessary. First, assume that the items are ordered such that $w_j \ge w_b$ for $j = 1, \ldots, b - 1$, and $w_j \le w_b$ for $j = b + 1, \ldots, n$, which is obtainable in time $O(n)$ by using a partitioning technique. The ordering means that each time we add an item $t \ge b$ to a feasible solution, at most one item $s < b$ needs to be removed in order to maintain feasibility, meaning that the values μ in lines 6–8 may be considered in an arbitrary order within each line. We only need to distinguish between states with $\mu \le C$ and $\mu > C$, by keeping them in two separate sets.

4. 0–1 KNAPSACK PROBLEM

Assume that the items are ordered according to nonincreasing profit-to-weight ratios p_j/w_j, and define the break item by $b = \min\{j : \sum_{i=1}^{j} w_i > C\}$. The profit and weight sum of the break solution x' is $\bar{p}$ and $\bar{w}$. Invariant (3) now becomes

$$s_t(\mu, \pi) = \max s \begin{cases} \text{there exists a balanced filling } x \text{ which satisfies} \\ \sum_{j=1}^{s-1} p_j + \sum_{j=s}^{t} p_j x_j = \pi \\ \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j = \mu \\ x_j \in \{0, 1\}, \quad j = s, \ldots, t, \end{cases} \tag{4}$$

where we set $s_t(\mu, \pi) = 0$ if no balanced solution exists. For $t = b - 1$ only one value of $s_t(\mu, \pi)$ is positive, namely $s_t(\overline{w}, \overline{p}) = b$, as only the break solution is balanced initially.

The magnitude of the coefficients is $W = \max_{j=1,\ldots,n} w_j$ (resp. $P = \max_{j=1,\ldots,n} p_j$). As a consequence of the balanced operations we have $c - W < \mu \leq c + W$, and to derive a similar constraint on π we apply some bounding rules. The Dantzig upper bound on (1), which appears by continuous relaxation, is given by $u = \lfloor \overline{p} + (C - \overline{w})p_b/w_b \rfloor$. An upper bound on a state (μ, π) may be derived by using the Dembo and Hammer [6] upper bound $\tilde{u}(\mu, \pi) = \pi + \lfloor (C - \mu)p_b/w_b \rfloor$. Obviously, an upper bound on a state cannot be better than the upper bound on (1) since we have fixed some decision variables in the state, thus we have $\tilde{u}(\mu, \pi) \leq u$, i.e.,

$$\pi \leq \beta(u) = u - \lfloor (C - \mu)p_b/w_b \rfloor. \tag{5}$$

In addition, any state (μ, π) can be discarded if its upper bound is worse than a given incumbent solution value z. Thus, any live state must satisfy $\tilde{u}(\mu, \pi) = \pi + \lfloor (C - \mu)p_b/w_b \rfloor \geq z + 1$, implying

$$\pi \geq \alpha(\mu) = z + 1 - \lfloor (C - \mu)p_b/w_b \rfloor. \tag{6}$$

At any stage we have $\alpha(\mu) \leq \pi \leq \beta(\mu)$, so the number of profit sums π corresponding to a weight sum μ can at most be $\beta(\mu) - \alpha(\mu) + 1 = u - z = g$, where $g \leq P$ as we may use $z = \overline{p}$ as incumbent solution, and obviously $u < \overline{p} + p_b$. This leads to the generalized algorithm:

```
1.  ALGORITHM balknap.
2.  for μ ← C − W + 1 to C do
3.    for π ← α(μ) to β(μ) do s_{b−1}(μ, π) ← 0;
4.  for μ ← C + 1 to C + W do
5.    for π ← α(μ) to β(μ) do s_{b−1}(μ, π) ← 1;
6.  s_{b−1}(w̄, p̄) ← b;
7.  for t ← b to n do
8.    for μ ← C − W + 1 to C + W do
9.      for π ← α(μ) to β(μ) do s_t(μ, π) ← s_{t−1}(μ, π);
10.   for μ ← C − W + 1 to C do
11.     for π ← α(μ) to β(μ) do
12.       μ' ← μ + w_t; π' ← π + p_t;
          s_t(μ', π') ← max{s_t(μ', π'), s_{t−1}(μ, π)};
13.   for μ ← C + w_t downto C + 1 do
14.     for π ← α(μ) to β(μ) do
15.       for j ← s_{t−1}(μ, π) to s_t(μ, π) − 1 do
16.         μ' ← μ − w_j; π' ← π − p_j;
            s_t(μ', π') ← max{s_t(μ', π'), j}
```

The time and space complexity is $O(nWg)$ which easily may be verified as in Proposition 3. Since $g \leq P$ the complexity may be written $O(nWP)$. Furthermore, the tighter incumbent solution z we provide, the faster solution times we get. In Pisinger [13 p. 86] it is shown that the gap $g = u - z$ typically is of magnitude 0−10 for large-sized instances from the literature.

Note that a complete ordering of the items according to nonincreasing profit-to-weight ratios is not necessary, as we only need the break item b for deriving upper and lower bounds. The break item may be found in linear time through partitioning [3].

5. OTHER GENERALIZATIONS

In the Multiple-choice Subset-sum Problem (MCSSP) we have k classes; each class N_i contains weights $w_{1i}, \ldots, w_{ni}$. The problem is to select one weight from each class such that the total weight sum is maximized without exceeding the capacity C. Thus, we have

$$\begin{aligned} \text{maximize} \quad & z = \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \leq C, \\ & \sum_{j \in N_i} x_{ij} = 1, \quad i = 1, \ldots, k, \\ & x_{ij} \in \{0,1\}, \quad i = 1, \ldots, k, j \in N_i \end{aligned} \tag{7}$$

where $x_{ij} = 1$ if weight j was chosen in class N_i. Let $\alpha_i = \arg\min_{j \in N_i}\{w_{ij}\}$ and $\beta_i = \arg\max_{j \in N_i}\{w_{ij}\}$ for $i = 1, \ldots, k$ be the indices of the smallest and largest weights in each class i. The break class b is defined by $b = \min\{j : \sum_{i=1}^{j} w_{i\beta_i} > C - \sum_{i=j+1}^{k} w_{i\alpha_i}\}$. With these definitions, the `balsub` algorithm may be generalized to

1. ALGORITHM `balmcsub`.
2. **for** $\mu \leftarrow C - W + 1$ **to** C **do** $s_{b-1}(\mu) \leftarrow 0$;
3. **for** $\mu \leftarrow C + 1$ **to** $C + W$ **do** $s_{b-1}(\mu) \leftarrow 1$;
4. $s_{b-1}(\overline{w}) \leftarrow b$;
5. **for** $t \leftarrow b$ **to** k **do**
6. **for** $\mu \leftarrow C - W + 1$ **to** $C + W$ **do** $s_t(\mu) \leftarrow s_{t-1}(\mu)$;

```
7.    for μ ← C − W + 1 to C do
8.       for i ∈ N_t do μ' ← μ + w_ti − w_tα_t;
         s_t(μ') ← max{s_t(μ'), s_{t−1}(μ)};
9.    for μ ← C + w_t downto C + 1 do
10.      for j ← s_{t−1}(μ) to s_t(μ) − 1 do
11.         for i ∈ N_j do μ' ← μ + w_ji − w_jβ_j; s_t(μ') ← max{s_t(μ'), j};
```

The algorithm runs in $O(W\sum_{i=1}^{k} n_i) = O(nW)$; thus, we have the same low complexity as for SSP.

The MCSSP problem may be used for tightening constraints in several IP applications which have a multiple-choice constraint. For a concrete application of constraint tightening see Pisinger [16]. MCSSP may also (partially) be used for solving the Multiple-choice Knapsack Problem: Using the principle of Balas and Zemel [3], a *core* is selected containing items from each class with appropriate profit-to-weight ratios. Then the core problem is solved as a MCSSP, yielding a lower bound. In a similar way as in [3] it may be proved that this lower bound corresponds to the optimal solution with a very high probability.

The Bounded Knapsack Problem (BKP) is a generalization of KP, where m_j items of each item type j are available; thus the last constraint in (1) should be replaced by $x_j \in \{0, 1, \ldots, m_j\}$, $j = 1, \ldots, n$.

Martello and Toth [11] present different techniques for transforming BKP to an ordinary KP. As our main goal is to keep the coefficient sizes as small as possible, the best transformation is to handle all m_j items of each type as individual items. The obtained KP with $\sum_{j=1}^{n} m_j$ variables is solved using the `balknap` algorithm in time $O(WP\sum_{j=1}^{n} m_j) \le O(nWPM)$, when $w_j \le W$, $p_j \le P$ and $m_j \le M$, which is linear provided the coefficients are bounded by constants. This should be compared to a basic recursion as presented in [11], which has solution times of $O(C\sum_{j=1}^{n} m_j) \le O(n^2WM)$.

For Unbounded Knapsack Problems balancing is less attractive. Gilmore and Gomory [7] gave a recursion running in time $O(nC)$ corresponding to $O(n^2W)$. If we use balancing, we may transform the problem to a Bounded Knapsack Problem by introducing the bounds $m_j = c/w_j \le nW$ on each type. Thus, the complexity becomes $O(n^2WP)$.

It is obvious that balancing may be applied to other problems from the knapsack family, leading to new pseudo polynomial time bounds. Balancing has been applied for solving KP through branch-and-bound by Pisinger [15], leading to reasonable solution times for several of the considered problems. The worst-case solution time is, however, not improved by this technique. A dynamic programming algorithm based on similar principles as balancing has recently been used to solve Strongly Correlated Knapsack Problem orders of magnitude faster than previous approaches [17].

Several fully polynomial approximation schemes for knapsack-like problems are based on state-space relaxation of a dynamic programming algorithm. The presented balanced algorithms may lead to new approximation algorithms.

There is no obvious way how balancing can be applied in connection with the partitioning technique, as when an optimal solution is partitioned the two parts need not be balanced at all. However, it is possible to obtain better time bounds by combining the two approaches, such that partitioning is applied for the large weighted items and balanced dynamic programming is used for the small weighted items. For the SSP assume that the weights are ordered according to nonincreasing weights and let d be defined by $d = \min_{j=0,2,\ldots,n}\{j : 2^{n/2}(n-j)w_{j+1} < 2^{j/2+1}(n-j-2)w_{j+3}\}$. Now use partitioning to enumerate items $j = 1,\ldots,d$, and for each state obtained by merging the two sets, apply `balsub` to enumerate items $j = d+1,\ldots,n$. The time bound of this approach becomes $O(2^{d/2}(n-d)w_{d+1}) \le O(\min\{2^{n/2}, nW\})$.

6. COMPUTATIONAL EXPERIMENTS

We have restricted the computational experiments to the SSP, as the time bound of `balsub` is the most attractive, compared to previous techniques. Five types of data instances presented in Martello and Toth [11] are considered:

`P(3)`:	w_j randomly distributed in $[1, 10^3]$, and $C = \lfloor n10^3/4 \rfloor$.
`P(6)`:	w_j randomly distributed in $[1, 10^6]$, and $C = \lfloor n10^6/4 \rfloor$.
`even/odd`:	w_j even, randomly distributed in $[1, 10^3]$, and $C = 2\lfloor n10^3/8 \rfloor + 1$ (odd).
`avis`:	$w_j = n(n+1) + j$, and $C = n(n+1)\lfloor (n-1)/2 \rfloor + n(n-1)/2$.
`todd`:	set $k = \lfloor \log_2 n \rfloor$, then $w_j = 2^{k+n+1} + 2^{k+j} + 1$, and $C = \lfloor \frac{1}{2}\sum_{j=1}^{n} w_j \rfloor$.

Jereslow [10] showed that every branch-and-bound algorithm enumerates an exponentially growing number of nodes when solving `even/odd` problems. Avis [4] showed that any recursive algorithm which does not use dominance will perform poorly for the `avis` problems. Finally, Todd [4] constructed the `todd` problems such that any algorithm which uses upper bounding tests, dominance relations, and rudimentary divisibility arguments will have to enumerate an exponential number of states. The running times of three different approaches are compared in Table 1: The `bellman` recursion [2], the `balsub` algorithm, and finally, the `mtsl`

TABLE 1

Algorithm	n	P(3)	P(6)	even/odd	avis	todd
	10	0.00	0.00	0.00	0.00	0.00
	30	0.02	—	0.01	0.01	—
	100	0.33	—	0.21	1.57	—*
	300	4.11	—	4.84	—	—*
bellman	1000	52.86	—	69.34	—	—*
	3000	505.38	—	723.25	—*	—*
	10000	—	—*	—	—*	—*
	30000	—	—*	—	—*	—*
	100000	—	—*	—	—*	—*
	10	0.00	0.00	0.00	0.00	0.00
	30	0.00	0.01	3.84	12.39	0.00
	100	0.00	0.00	—	—	—*
	300	0.00	0.00	—	—	—*
mtsl	1000	0.00	0.00	—	—	—*
	3000	0.00	0.01	—	—*	—*
	10000	0.00	—*	—	—*	—*
	30000	0.00	—*	—	—*	—*
	100000	0.02	—*	—	—*	—*
	10	0.00	5.37	0.00	0.00	0.12
	30	0.00	8.68	0.01	0.01	—
	100	0.00	4.21	0.02	0.26	—*
	300	0.00	2.62	0.07	15.21	—*
balsub	1000	0.00	2.12	0.22	562.38	—*
	3000	0.00	2.11	0.66	—*	—*
	10000	0.00	—*	2.22	—*	—*
	30000	0.00	—*	6.66	—*	—*
	100000	0.02	—*	23.76	—*	—*

Note. Solution times in seconds, as average of 100 instances (HP9000/730). Entries marked with an asterik could not be generated at the present computer due to limits on the size of integers. The time limit was set to 20 h for each series of instances and a ''—'' indicates that the instances could not be solved within this limit.

algorithm by Martello and Toth [11] which is a branch-and-bound algorithms using partial dynamic programming enumeration. The `mtsl` algorithm was obtained from [11], while the `balsub` algorithm is available from `http://www.diku.dk/~pisinger/codes.html`.

For the randomly distributed problems `P(3)` and `P(6)` we have W bounded by a (large) constant. Thus the `bellman` recursion runs in $O(n^2)$ time, while `balsub` has linear solution time. The problems `P(3)` and `P(6)` have the property that several solutions to $\sum_{j=1}^{n} w_j x_j = C$ do exist

when n is large; thus generally, `balsub` may terminate before a complete enumeration. The `bellman` recursion has to enumerate all states up to at least $t = b$ before it can terminate. The `mtsl` algorithm is the fastest for these problems, since the branch-and-bound technique quickly finds a solution satisfying $\sum_{j=1}^{n} w_j x_j = C$.

For the `even/odd` problems no solution satisfying $\sum_{j=1}^{n} w_j x_j = C$ exists, meaning that we reach the worst-case solution times the number of algorithms. Here `balsub` runs in $O(nW)$ and, thus, linear time. The `bellman` recursion has complexity $O(n^2W)$, and thus, cannot solve problems larger than $n = 3000$. Finally `mtsl` is not able to solve problems larger than $n = 30$.

The `avis` problems have weights of magnitude $O(n^2)$ while the capacity is of magnitude $O(n^3)$, so the `bellman` recursion demands $O(n^4)$ time, while `balsub` solves the problem in $O(n^3)$. Algorithm `mtsl`, again, cannot solve problems larger than $n = 30$.

Finally, the `todd` problems are considered. As expected, none of the algorithms are able to solve more than tiny instances due to the exponentially growing weights.

7. CONCLUSION

We have presented a new technique for deriving tight pseudo polynomial time bounds for Knapsack Problems. Concrete algorithms have been presented for the SSP, KP, BKP and MCSSP, where the solution times are $O(nW)$, $O(nWP)$, $O(nWPM)$, and $O(nW)$, respectively. When all coefficients are bounded by a constant, these solution times are linear. Due to the larger constants for KP and BKP, the approach is less attractive for these problems than for the two variants of SSP, but still the algorithms presented may be applicable for very large-sized problems having moderate coefficient sizes. Smaller sized problems may also be solved efficiently by `balknap` in those cases where a good lower bound is easy to derive but optimality is difficult to prove.

The time bounds fully describe the nature of Knapsack Problems, as it clearly states that all the complexity is hidden in the magnitude of the coefficients. This conforms with the observation by Chvátal [4] who had to use exponentially growing weights in order to construct hard instances of KP.

The computational experiments with SSP algorithms have demonstrated that even very hard instances of large size may be solved in reasonable time by using `balsub`; thus, the results presented are also important from a practical point of view.

REFERENCES

1. A. Andersson, Faster deterministic sorting and searching in linear space, *in* "Proceedings of the 37th Symposium on foundations of computer science (FOCS'96)," pp. 135–141.
2. R. E. Bellman, "Dynamic Programming," Princeton Univ. Press, Princeton, NJ, 1957.
3. E. Balas and E. Zemel, An algorithm for large zero–one knapsack problems, *Oper. Res.* **28** (1980), 1130–1154.
4. V. Chvátal, Hard knapsack problems, *Oper. Res.* **28** (1980), 1402–1411.
5. T. H. Cormen, C. E. Rivest, and R. L. Rivest, "Introduction to Algorithms," MIT Press, Cambridge, MA, 1989.
6. R. S. Dembo and P. L. Hammer, A Reduction algorithm for knapsack problems, *Methods Oper. Res.* **36** (1980), 49–60.
7. P. C. Gilmore and R. E. Gomory, Multi-stage cutting stock problems of two and more dimensions, *Oper. Res.* **13** (1965), 94–120.
8. E. Horowitz and S. Sahni, Computing partitions with applications to the knapsack problem, *J. Assoc. Comput. Mach.* **21** (1974), 277–292.
9. T. Ibaraki, Enumerative approaches to combinatorial optimization—Part 2, *Ann. Oper. Res.* **11** (1987), 376–388.
10. R. G. Jeroslow, Trivial integer programs unsolvable by branch-and-bound, *Math Programming* **6** (1974), 105–109.
11. S. Martello and P. Toth, "Knapsack Problems: Algorithms and Computer Implementations," Wiley, Chichester, England, 1990.
12. K. Mehlhorn and S. Näher, Bounded ordered dictionaries in $O(\log \log n)$ time and $O(n)$ space, *Inform. Process. Lett.* **35** (1990), 183–189.
13. D. Pisinger, Algorithms for knapsack problems, Ph.D. thesis, DIKU, University of Copenhagen, 1995. [Report 95/1. Available from http://www.diku.dk]
14. D. Pisinger, A minimal algorithm for the multiple-choice knapsack problem, *Europ. J. Oper. Res.* **83** (1995), 394–410.
15. D. Pisinger, An expanding-core algorithm for the exact 0–1 knapsack problem, *Europ. J. Oper. Res.* **87** (1995), 175–187.
16. D. Pisinger, An exact algorithm for large multiple knapsack problems, *Europ. J. Oper. Res.* **114** (1999), 528–541.
17. D. Pisinger, A fast algorithm for strongly correlated knapsack problems, *Discrete Appl. Math.* **89** (1998), 197–212.