An $O(nr)$ algorithm for the Subset-sum Problem

– and other balanced knapsack algorithms

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

Abstract

A new technique called *balancing* is presented for the solution of Knapsack Problems. It is proved that an optimal solution to KP 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 coefficients are bounded by a constant.

Extensive computational experiments are presented to document that the derived algorithms also in practice solve several difficult problems from the literature faster than previous approaches. It is discussed how the presented enumeration schemes may affect approximate algorithms for the problems considered, and how balancing may be combined with other techniques known for Knapsack Problems in order to obtain tighter time bounds.
Keywords: Knapsack Problem, Dynamic Programming.

1 Introduction

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

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

where $p_j, w_j > 0$ are the profits and weights of the items, while c is the capacity of the knapsack. To avoid trivial cases we assume $\sum_{j=1}^{n} w_j > c$, and $w_j \le c$ for $j = 1,\dots,n$.

Technical Report 95/6, DIKU, University of Copenhagen, Denmark

Branch-and-bound algorithms for Knapsack Problems have attained much interest during the last two decades due to their ability to solve several practically occuring problems. It is however not possible to give reasonable time bounds for these algorithms, as instances may be constructed which demand a complete enumeration. If better time bounds are demanded, two techniques appear in the literature: *Dynamic programming*, introduced by Bellman [1], allows us to solve a KP in *pseudo polynomial* time $O(nc)$, which for moderate capacities lead to attractive solution times. If the capacity however is very large, Horowicz and Sahni [6] introduced the *partitioning* technique: The items are partitioned in two sets which each are 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 need to consider balanced solutions, tighter time bounds than previously seen may be derived. Among the main results should be emphasized that — provided that 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 Bounded Knapsack 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 is able to solve most difficult problems.

2 Balanced operations

First we need some definitions: 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 which occur 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 follows:

- The break solution x' is a balanced filling.
- *Balanced insert:* If we have a balanced filling x with $\sum_{j=1}^{n} w_j x_j \le c$ and change a variable x_t, $(t \ge b)$ 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 $s_1, \dots, s_\alpha$ be the indices $s_i < b$ where $x^*_{s_i} = 0$, and $t_1, \dots, t_\beta$ 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 $\{s_1, \dots, s_\alpha\}$ and $\{t_1, \dots, t_\beta\}$ 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 however implies that x^* could not be a feasible solution from the start as the knapsack is filled and we still have to insert items.

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, as a better feasible solution can be obtained by *not* removing the remaining items s_i. □

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: $c - r < \sum_{j=1}^{n} w_j x_j \leq c + r$, where the *range* r is given by $r = \max_{j=1}^{n} w_j$.

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

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

				b			
j	1	2	3	4	5	6	$c = 15$
w_j	6	4	2	6	4	3	

μ	$s_{b-1}(\mu)$		$s_b(\mu)$	$s_{b+1}(\mu)$	$s_{b+2}(\mu)$	
10	0		**1**	**1**	**1**	
11	0		0	0	**1**	
12	**4**		**4**	**4**	**4**	
13	0		0	0	**2**	
14	0		**2**	**3**	**3**	
15	0	c	0	0	**4**	c
16	1		**3**	**4**	**4**	
17	1		1	1	**3**	
18	1		**4**	**4**	**4**	
19	1		1	1	1	
20	1		1	1	1	
21	1		1	1	1	

Figure 1: The items and table $s_t(\mu)$ for a given instance.

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 $x_s, \dots, x_t$, applying the following (unusual) dominance relation:

Definition 2 Given two states (s, t, μ) and (s', t', μ'). If $\mu = \mu'$, $s \geq s'$ 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 fathom 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 $x_s, \dots, x_t$. Therefore let $s_t(\mu)$ for $t = b-1, \dots, n$ and $c - r < \mu \leq c + r$ be defined as

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

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

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 $\mu \leq c$ need to be saved, but in order to improve efficiency, we use $s_t(\mu)$ for $\mu > c$ to memoize that items $j < s_t(\mu)$ have been removed once before. We get:

Algorithm `balsub`
for $\mu \leftarrow c - r + 1$ **to** c **do** $s_{b-1}(\mu) \leftarrow 0$;
for $\mu \leftarrow c + 1$ **to** $c + r$ **do** $s_{b-1}(\mu) \leftarrow 1$;
$s_{b-1}(\overline{w}) \leftarrow b$;

5 **for** $t \leftarrow b$ **to** n **do**
6   **for** $\mu \leftarrow c - r + 1$ **to** $c + r$ **do** $s_t(\mu) \leftarrow s_{t-1}(\mu)$;
7   **for** $\mu \leftarrow c - r + 1$ **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 Figure 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 $s_t(\mu)$, $\mu > c$ is used for memoizing, states with $\mu > c$ are set to $s_t(\mu) = 1$ as no items $j < s_t(\mu)$ have ever been removed.

Now we consider the items $t = b, \ldots, n$ in line 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 add 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.

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 $\mu' < \mu$, we must consider the weights μ in decreasing order in line 8 in order to allow multiple removals. This ensures that all states will be feasible after each iteration of t.

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 memoizing 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 $O(nr)$ in time and space.

Proof *Space:* The array $s_t(\mu)$ has size $(n - b + 1)(2r)$, thus $O(nr)$. *Time:* Lines 2-3 demand $2r$ operations. Line 6 is executed $2r(n-b+1)$ times. Line 7 is executed $r(n-b+1)$ times. Finally, for each $\mu > c$, line 9 is executed totally $s_n(\mu) \leq b$ times. Thus during the whole process, line 9 is executed at most rb times. This proves the stated. □

The disadvantage to `balsub` is, that we use dynamic programming by *pulling*, meaning that all $2r$ states are considered at each iteration. An algorithm based on dynamic programming by *reaching* (see Ibaraki for definitions [7]) may be obtained by only considering those states where $s_t(\mu) \neq 0$ for $\mu \leq c$ and $s_t(\mu) \neq 1$ for $\mu > c$. A data structure should be chosen which supports 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 $w_j \geq w_b$ for $j = 1, \ldots, b-1$, and $w_j \leq w_b$ for $j = b+1, \ldots, n$, which is obtainable in time $O(n)$, e.g. by using the `partsort` algorithm by Pisinger [12]. The ordering means, that each time we add an item $t \geq b$ to a feasible solution, at most one item $s < b$ need to be removed in order to maintain feasibility, meaning that the values μ may be considered in an arbitrary order at each stage of t. We only need to partition the states in those with $\mu \leq c$ and those with $\mu > c$, while hashing may be used to access the states inside the loops.

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 $\overline{p}$ and $\overline{w}$, and the Dantzig upper bound is given by $u = \lfloor \overline{p} + (c - \overline{w})p_b/w_b \rfloor$. 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 *range* of the coefficients is given as $r_1 = \max_{j=1,\ldots,n} w_j$ resp. $r_2 = \max_{j=1,\ldots,n} p_j$. As a consequence of the balanced operations we have $c - r_1 < \mu \leq c + r_1$, and to derive a similar constraint on π we apply some bounding rules. Since the Dembo and Hammer [4] upper bound $\tilde{u}$ is weaker than the Dantzig upper bound u we have $\tilde{u}(\mu, \pi) = \pi + \lfloor (c - \mu)p_b/w_b \rfloor \leq u$, thus

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

On the other hand, if a state (μ, π) does not have an upper bound better than the current solution z, then it may be fathomed. Thus any live state must satisfy $\tilde{u}(\mu, \pi) = \pi + \lfloor (c - \mu)p_b/w_b \rfloor \geq z + 1$ meaning that

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

Thus at any stage we have $\alpha(\mu) \leq \pi \leq \beta(\mu)$, meaning that the number of profit sums π corresponding to a weight sum μ can at most be $g = u - z$, where $g \leq r_2$ as we may use the lower bound $z = \overline{p}$. This leads to the following generalized algorithm:

Algorithm `balknap`
for $\mu \leftarrow c - r_1 + 1$ **to** c **do**
  for $\pi \leftarrow \alpha(\mu)$ **to** $\beta(\mu)$ **do** $s_{b-1}(\mu, \pi) \leftarrow 0$;
for $\mu \leftarrow c + 1$ **to** $c + r_1$ **do**
  for $\pi \leftarrow \alpha(\mu)$ **to** $\beta(\mu)$ **do** $s_{b-1}(\mu, \pi) \leftarrow 1$;
$s_{b-1}(\overline{w}, \overline{p}) \leftarrow b$;
for $t \leftarrow b$ **to** n **do**
  for $\mu \leftarrow c - r_1 + 1$ **to** $c + r_1$ **do**
    for $\pi \leftarrow \alpha(\mu)$ **to** $\beta(\mu)$ **do** $s_t(\mu, \pi) \leftarrow s_{t-1}(\mu, \pi)$;
  for $\mu \leftarrow c - r_1 + 1$ **to** c **do**
    for $\pi \leftarrow \alpha(\mu)$ **to** $\beta(\mu)$ **do**
      $\mu' \leftarrow \mu + w_t;\ \pi' \leftarrow \pi + p_t;\ s_t(\mu', \pi') \leftarrow \max\{s_t(\mu', \pi'),\ s_{t-1}(\mu, \pi)\}$;
  for $\mu \leftarrow c + w_t$ **downto** $c + 1$ **do**
    for $\pi \leftarrow \alpha(\mu)$ **to** $\beta(\mu)$ **do**
      for $j \leftarrow s_{t-1}(\mu, \pi)$ **to** $s_t(\mu, \pi) - 1$ **do**
        $\mu' \leftarrow \mu - w_j;\ \pi' \leftarrow \pi - p_j;\ s_t(\mu', \pi') \leftarrow \max\{s_t(\mu', \pi'),\ j\}$;

The time and space complexity is $O(nr_1g)$ which easily may be verified as in Proposition 3. Since $g \leq r_2$ the complexity may be written $O(nr_1r_2)$. Notice that the tighter lower bound z we provide as initial solution, the faster solution times we get. As most Knapsack Problems occuring in the literature have $g = u - z \ll r_2$ we may expect solution times comparable to those of `balsub`.

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 a partitioning algorithm.

5 Other generalizations

In the Multiple-choice Subset-sum problem (MCSSP) we have k classes, each class N_i containing weights $w_{1i}, \ldots, w_{n_i i}$. 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

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

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 weight in each class i. The break class b is defined by $b = \min\{j : \sum_{i=1}^{j} w_{i\beta_i} > c - \sum_{i=j+1}^{k} w_{i\alpha_i}\}$. We get the straightforward generalization:

Algorithm `balmcsub`
for $\mu \leftarrow c - r + 1$ **to** c **do** $s_{b-1}(\mu) \leftarrow 0$;
for $\mu \leftarrow c + 1$ **to** $c + r$ **do** $s_{b-1}(\mu) \leftarrow 1$;
$s_{b-1}(\overline{w}) \leftarrow b$;
for $t \leftarrow b$ **to** k **do**
 for $\mu \leftarrow c - r + 1$ **to** $c + r$ **do** $s_t(\mu) \leftarrow s_{t-1}(\mu)$;
 for $\mu \leftarrow c - r + 1$ **to** c **do**
  for $i \in N_t$ **do** $\mu' \leftarrow \mu + w_{ti} - w_{t\alpha_t}$; $s_t(\mu') \leftarrow \max\{s_t(\mu'),\ s_{t-1}(\mu)\}$;
 for $\mu \leftarrow c + w_t$ **downto** $c + 1$ **do**
  for $j \leftarrow s_{t-1}(\mu)$ **to** $s_t(\mu) - 1$ **do**
   for $i \in N_j$ **do** $\mu' \leftarrow \mu + w_{ji} - w_{j\beta_j}$; $s_t(\mu') \leftarrow \max\{s_t(\mu'),\ j\}$;

The algorithm runs in $O(r \sum_{i=1}^{k} n_i) = O(nr)$, thus we have the same low computational effort as for subset-sum problem.

The MCSSP problem may be used for tightening constraints in several IP applications which have a multiple-choice constraint. For a concrete application of this tightening see Pisinger [13]. The problem may also in some extent be used for solving a Multiple-choice Knapsack problem. As Balas and Zemel [2] showed for the 0-1 Knapsack Problem, one may derive a core of the problem, solve it as a Subset-sum problem, and use this as a (good!) lower bound. Actually this lower bound may be proved to be optimal with a very large probability. For the Multiple-choice Knapsack Problem we define a core as a number of items in each class, where the gradient to these items is sufficiently close to the gradient of the break class. See Pisinger [11] for a discussion of cores for Multiple-choice Knapsack Problems.

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 $x_j \in \{0, 1, \ldots, m_j\}, \quad j = 1, \ldots, n$.

Martello and Toth [10] 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 $O(r_1 r_2 \sum_{j=1}^{n} m_j) \leq O(n r_1 r_2 r_3)$, when $w_j \leq r_1$, $p_j \leq r_2$ and $m_j \leq r_3$, which is linear for all problems with coefficients distributed in a bounded range. This should be compared to a basic recursion as presented in [10], which has solution times of $O(c \sum_{j=1}^{n} m_j) \leq O(nc^2)$ which for bounded coefficient sizes correspond to $O(n^2 r^2)$.

For Unbounded Knapsack Problems balancing is less attractive. Gilmore and Gomory [5] gave a recursion running in time $O(nc)$ corresponding to $O(n^2 r_1)$. If we use balancing, we may transform the problem to a Bounded Knapsack Problem by introducing the bounds $m_j = c/w_j \leq n r_1$ on each type. Thus the complexity becomes $O(n^2 r_1 r_2)$.

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 [12] 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 Problems thousinds of times faster than previous approaches [14].

Several fully polynomial approximations schemes for knapsack-like problems are based on state-space relaxation of a dynamic programming algorithm. With the improved time bounds for several of these problems, we immediately get improved time bounds for these approximation schemes.

There is no obvious way how balancing may be applied in connection with the partitioning technique, as when an optimal solution is partitioned, the two parts need not be balanced at all. It is however 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 $d = \min_{j=0,2,\ldots,n}\{j : 2^{j/2}(n-j)w_{j+1} < 2^{j/2+1}(n-j-2)w_{j+3}\}$. 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 $O(2^{d/2}(n-d)w_{d+1}) \leq O(\min\{2^{n/2}, nr\})$.

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 [10] 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 $c = \lfloor \frac{1}{2} \sum_{j=1}^{n} w_j \rfloor$.

Jeroslow [8] showed that every branch-and-bound algorithm enumerates an exponentially growing number of nodes when solving `even/odd` problems. Avis [3] showed that any recursive algorithm which does not use dominance will perform poorly for the `avis` problems. Finally Todd [3] 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 I: The `bellman` recursion [1], the `balsub` algorithm,

Table I: Solution times in seconds, as average of 100 instances (hp9000/730). Entries marked with an asterisk could not be generated at the present computer.

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	–*	–*

and finally the `mtsl` algorithm by Martello and Toth [10] which is a branch-and-bound algorithms using partial dynamic programming enumeration.

For the randomly distributed problems `P(3)` and `P(6)` we have r 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 however the fastest for these problems, as 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 $\sum_{j=1}^{n} w_j x_j = c$ do exist, meaning that we get a strict $O(nr)$ and thus linear solution time for `balsub`. The `bellman` recursion has complexity $O(n^2 r)$, 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. Due to the exponentially growing weights, none of the algorithms are able to solve more than tiny instances, as could be expected.

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 $O(nr)$, $O(nr^2)$, $O(nr^3)$ and $O(nr)$ respectively. When all coefficients are bounded by a constant r, these solution times are linear. Due to the huge constants for KP and BKP, the approach is less attractive for these problems than for the two varients 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 [3] 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 presented results are also important from a practical point of view.

References

[1] R. E. Bellman (1957), *Dynamic programming*, Princeton University Press, Princeton, NJ.

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

[3] V. Chvátal (1980), "Hard Knapsack Problems", *Operations Research*, **28**, 1402–1411.

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

[5] P.C. Gilmore and R.E. Gomory (1965), "Multi-stage cutting stock problems of two and more dimensions", *Operations Research*, **13**, 94–120.

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

[7] T. Ibaraki (1987), "Enumerative Approaches to Combinatorial Optimization – Part 2", *Annals of Operations Research*, **11**, 376–388.

[8] R. G. Jeroslow (1974), "Trivial Integer Programs Unsolvable by Branch-and-Bound", *Mathematical Programming*, **6**, 105–109.

[9] S. Martello and P. Toth (1984), "A mixture of dynamic programming and branch-and-bound for the subset-sum problem", *Management Science*, **30**, 765–771.

[10] S. Martello and P. Toth (1990), *Knapsack Problems: Algorithms and Computer Implementations,* Wiley, Chichester, England.

[11] D. Pisinger (1995), "A minimal algorithm for the Multiple-choice Knapsack Problem," *European Journal of Operational Research*, **83**, 394–410.

[12] D. Pisinger (1995), "An expanding-core algorithm for the exact 0-1 knapsack problem", *European Journal of Operational Research*, **87**, 175–187.

[13] D. Pisinger (1995), "The Multiple Loading Problem", *Proceedings NOAS'95*, University of Reykjavík, Iceland, August 18–19, 1995.

[14] D. Pisinger (1996), "Strongly correlated Knapsack Problems are trivial to solve", *Symposium on Combinatorial Optimisation CO'96*, Imperial College of Science, London, March 27–29, 1996.