Filtering Bounded Knapsack Constraints in Expected Sublinear Time*

Yuri Malitsky, Meinolf Sellmann
Brown University, PO Box 1910
Providence, RI 02912
{ynm,sello}@cs.brown.edu

Radoslaw Szymanek
Ecole Polytechnique Federale de Lausanne
INR (Batiment IN), 1015 Lausanne, Switzerland
radoslaw.szymanek@gmail.com

Abstract

We present a highly efficient incremental algorithm for propagating bounded knapsack constraints. Our algorithm is based on the sublinear filtering algorithm for binary knapsack constraints by Katriel et al. and achieves similar speed-ups of one to two orders of magnitude when compared with its linear-time counterpart. We also show that the representation of bounded knapsacks as binary knapsacks leads to ineffective filtering behavior. Experiments on standard knapsack benchmarks show that the new algorithm significantly outperforms existing methods for handling bounded knapsack constraints.

Introduction

Linear resource constraints are important building blocks of many applications, including scheduling, configuration, and network design. Accordingly, knapsack has been the subject of intense research efforts in the optimization community. There exist highly efficient methods for solving pure knapsack problems (see, e.g., (Pisinger 2005)). The great practical importance of knapsack however is not as an optimization problem by itself, but as one constraint among many in a great variety of applications. Therefore, in constraint programming (CP) the knapsack constraint has been studied since 2000 (Fahle & Sellmann 2000). Efficient approximated and exact filtering algorithms have been developed (Fahle & Sellmann 2002; Sellmann 2003).

The most efficient filtering algorithm to date is from Katriel et al. (Katriel et al. 2007). It works in expected sublinear time and is designed exclusively for binary knapsack constraints. Using their work as a starting point, we generalize the method to handle the much more general case of bounded knapsack constraints. Through rigorous experiments we show that the new algorithm dramatically boosts filtering effectiveness over modeling bounded knapsacks using binary knapsack constraints. Moreover, we show that the algorithm works up to two orders of magnitude faster than existing methods for filtering bounded knapsack constraints.

*This work was supported in part by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

Filtering Binary Knapsack Constraints

Formal Background

Let us begin by reviewing the binary case.

Definition 1 (Binary Knapsack Problem (2KP)). *Given n items $X = \{x_1, \ldots, x_n\}$, profits $p_1, \ldots, p_n \in \mathbb{N}$, weights $w_1, \ldots, w_n \in \mathbb{N}$, and a capacity $C \in \mathbb{N}$, the binary knapsack problem consists in finding a subset of items $S \subseteq \{x_1, \ldots, x_n\}$ such that $\sum_{x_i \in S} w_i \leq C$ and the total profit $\sum_{x_i \in S} p_i$ is maximized.*

The choice of adding an item x_i to S is commonly modeled by a binary variable X_i that is set to 1 iff $x_i \in S$. In the linear relaxation of this representation, we are allowed to select fractions of items. Unlike the integer problem, which is NP-hard, the fractional problem is easy to solve (Dantzig 1957): First, we arrange the items in non-increasing order of efficiency, i.e., we assume that $p_1/w_1 \geq \cdots \geq p_n/w_n$. Then, we greedily insert the most efficient item, until doing so would exceed the capacity of the knapsack, i.e., until we have reached the item x_s such that

$$\sum_{j=1}^{s-1} w_j \leq C \qquad \text{and} \qquad \sum_{j=1}^{s} w_j > C.$$

We say that x_s is the *critical item* for our knapsack instance. We then select the maximum fraction of x_s that can fit into the knapsack, i.e., $C - \sum_{j=1}^{s-1} w_j$ weight units of x_s. The total profit for the fractional problem is then $\tilde{P} = \sum_{j=1}^{s-1} p_j + \frac{p_s}{w_s}(C - \sum_{j=1}^{s-1} w_j)$.

Definition 2 (Binary Knapsack Constraint). *Given a lower bound $B \in \mathbb{N}$, the (global) binary knapsack constraint enforces that $\sum_i w_i X_i \leq C$ and $\sum_i p_i X_i \geq B$, where the domain of each variable X_i is $D_i = \{0, 1\}$.*

Achieving generalized arc consistency for the constraint is NP-hard, but relaxed consistency with respect to the linear relaxation bound can be achieved in polynomial time (Fahle & Sellmann 2002): If an item taken in full lowers the profit for the fractional relaxation below B then this item cannot belong to an integral solution of quality B or better. If an item must belong to every fractional solution of profit at least B, then it must also belong to every integral solution of this quality.

We classify an item x_i as *mandatory* iff the fractional optimum of the instance $(X \setminus \{x_i\}, C)$ has profit less than B, i.e., when without x_i we cannot achieve a relaxed profit of at least B. Then, we remove the value 0 from D_i. On the other hand, we classify x_i as *forbidden* iff the optimal fractional profit of the instance $(X \setminus \{x_i\}, C - w_i)$ is less than $B - p_i$, i.e., if a solution that includes x_i cannot achieve a relaxed profit of B. Then, we remove value 1 from D_i. Clearly, the items in $\{x_1, \ldots, x_{s-1}\}$ are not forbidden and the items in $\{x_{s+1}, \ldots, x_n\}$ are not mandatory. Our task, then, is to determine which of $\{x_1, \ldots, x_s\}$ are mandatory and which of $\{x_s, \ldots, x_n\}$ are forbidden. From here on we consider the problem of identifying the mandatory items only as the case of forbidden items works analogously.

Filtering 2KPs in Amortized Linear Time

In (Sellmann 2003), as a by-product of an algorithm that achieves approximated ε-consistency for knapsack constraints, an algorithm was developed that can identify the mandatory items in $O(n \log n)$ time. More precisely, the algorithm runs in linear time plus the time it takes to sort the items by efficiency once at the root node. Thus, when the status of a number of items (to be included in or excluded from the knapsack) or the current profit bound or the capacity changes, the remaining items are already sorted and the computation can be re-applied in linear time. The total time spent on m invocations of the filtering algorithm on the same constraint is therefore $\Theta(n \log n + mn)$. We describe this algorithm below.

For each $i < s$, let $e_i = p_i / w_i$ be the efficiency of x_i. If we give up a total of $\tilde{w}_i$ weight of this efficiency (whereby we allow $\tilde{w}_i > w_i$), we lose $\tilde{w}_i e_i$ units of profit. On the other hand, we can insert $\tilde{w}_i$ units of weight using the items that are not part of the fractional knapsack solution.

We define $Q(w)$ to be the maximum profit obtainable with a total weight of at most w using the $1 - (C - \sum_{j=1}^{s-1} w_j)/w_s$ unused fraction of item x_s and the items $\{x_{s+1}, \ldots, x_n\}$. Then $\tilde{w}_i$ is chosen such that

$$\tilde{P} - \tilde{w}_i e_i + Q(\tilde{w}_i) = B. \quad (1)$$

Then, an item x_i is mandatory if and only if $\tilde{w}_i < w_i$.

The core observation in (Sellmann 2003) is the following: If, for two items x_i and x_j, we have that $e_i \geq e_j$, then $\tilde{w}_i \leq \tilde{w}_j$. Hence, if we traverse the items of $\{x_1, \ldots, x_s\}$ in that order (i.e., by non-increasing efficiency), then all $\tilde{w}_i$'s can be identified by a single linear scan of the items of $\{x_s, \ldots, x_n\}$, starting at x_s. That is, the computation of $\tilde{w}_1$ starts at the critical item and then, for each consecutive $\tilde{w}_i$, we begin our search at the last position which monotonically advances to the right. If we constantly keep track of the sum of weights and the sum of profits of all items up to the current position, we only need to spend linear time to determine all mandatory items.

Example: Consider the example in Table 1. The linear continuous relaxation inserts items x_1, x_2, x_3 fully, and 80% of x_4, which is the critical item. The value of the relaxation is $\tilde{P} = 9 + 3 + 12 + 0.8 * 5 = 28$, which is greater than $B = 25$. To determine whether x_1 is mandatory, we compute $\tilde{w}_1$. Clearly, we can afford to replace 1 weight unit of x_1 with the unused 20% of the critical item as $1(e_1 - e_4) = 2 \leq 3 = \tilde{P} - B$. However, we cannot afford to replace two additional weight units of x_1 with the help of x_5 as $1(e_1 - e_4) + 2(e_1 - e_5) = 2 + 5 > 3 = \tilde{P} - B$, so the computation of $Q(\tilde{w}_1)$ stops at x_5.

item	1	2	3	4	5	threshold
profit	9	3	12	5	1	$B = 25$
weight	3	1	6	5	2	$C = 14$
efficiency	3	3	2	1	0.5	$\tilde{P} = 28$

Table 1: Binary Knapsack Constraint.

We have now found that the new position is x_5. We compute $\tilde{w}_1$, which we already know is between 1 and 3: $\tilde{w}_1 = \frac{7}{5}$ since $1(e_1 - e_4) + 0.4(e_1 - e_5) = 2 + 1 = 3 = \tilde{P} - B$. Consequently, we cannot afford to lose more than 1.4 weight units of x_1 which weighs 3. And thus, x_1 is mandatory.

For x_2, we already know that $\tilde{w}_2 \geq \tilde{w}_1$, and we can therefore begin the search for the next position at x_5. Again, replacing additional weight using all weight units of item x_5 is infeasible as $1(e_2 - e_4) + 2(e_2 - e_5) = 2 + 5 > 3 = \tilde{P} - B$. In fact, as $e_2 = e_1$, we find that $\tilde{w}_2 = \tilde{w}_1 = \frac{7}{5}$. However, since x_2 only weighs 1, we can afford to lose this item completely, and it is not mandatory.

Finally, for x_3 we have that $\tilde{w}_3 \geq \tilde{w}_2 = \frac{7}{5}$. Again, we begin the search for the new position at the old one and find that replacing all weight units of x_5 is infeasible: $1(e_3 - e_4) + 2(e_3 - e_5) = 1 + 3 = 4 > 3 = \tilde{P} - B$. We find $\tilde{w}_3 = \frac{7}{3}$ as $1(e_3 - e_4) + \frac{4}{3}(e_3 - e_5) = 1 + 2 = 3 = \tilde{P} - B$. As the weight of x_3 is 6, we cannot afford to lose this item, and consequently determine its status as mandatory.

Note that the current position increases monotonically with $\tilde{w}_i$, and consequently the total effort for these computations is linear once the efficiency order is known.

Filtering 2KPs in Expected Sublinear Time

In (Katriel et al. 2007), an improved version of the previous algorithm was developed which is based on the following observation:

Lemma 1. *Let $x_i, x_j \in \{x_1, \ldots, x_{s-1}\}$ such that $e_i \geq e_j$ and $w_i \geq w_j$. If x_i is not mandatory, then x_j is not mandatory.*

This means that, if an item x_i is *not* mandatory, then we can skip over all following items x_j in the efficiency ordering that have a weight $w_j \leq w_i$. Since we know that $\tilde{w}_j \geq \tilde{w}_i \geq w_i$ we can even skip over all items x_j with weight $w_j \leq \tilde{w}_i$ as these cannot be mandatory either. This improvement was named *strong skipping* in (Katriel et al. 2007).

Based on this idea, we do not need to compute $\tilde{w}_i$ for all items x_i but only for some. Not much would be gained, though, if the search for the new position was conducted linearly as described before. Therefore, (Katriel et al. 2007) proposed computing the next position by binary search. This can be implemented efficiently using finger trees so that the

search effort is limited to the logarithm of the difference of the new and the old position (Katriel et al. 2007). Note that, in the equilibrium equation for $\tilde{w}_i$ (see Equation 1), the fractional contribution of the critical item is constant, and the contribution of the items between the critical item and the item at the current position is simply the sum profit of all items in between. Therefore, it is sufficient to maintain finger trees that store information about cumulative profits and weights of items sorted by decreasing efficiencies.

Then, after $O(n \log n)$-time preprocessing (to sort the items by efficiency), every repetition of the propagator requires $O(d \log n + l \log(n/l) + k \log(n/k))$ additional time, where d is the number of elements whose status was determined (potentially also by other constraints) to be mandatory since the last invocation of the filtering algorithm, $l \leq n$ is the number of items that are being filtered, and $k \leq n$ is the number of items that are considered but which are not mandatory. Note that k is bounded from above by the longest increasing and decreasing subsequences when considering the weights of the items sorted by their efficiencies. Note further that, if the weights of the items were independent of the efficiencies, we expect the lengths k of the longest increasing and decreasing subsequences to be in $O(\sqrt{n})$. For the more realistic case where weights and *profits* are uncorrelated, (Katriel et al. 2007) showed that the lengths k of the longest increasing and decreasing subsequences are expected to be in $O(n^{\frac{2}{3}})$.

Since the status of any element, represented by its binary variable, can be settled at most once on the path from the root to a leaf node in the search tree, the first two terms of the complexity are bounded by $O(n \log n)$ for the entire path. In total, m repetitions of the propagator down a path of the search tree take $O(n \log n + mk \log(n/k)) = O(n \log n + mn^{\frac{2}{3}} \log n)$ time. That is, the average call to the filtering algorithm costs $O(\frac{n \log n}{m} + n^{\frac{2}{3}} \log n)$ which is sublinear for sufficiently large $m \in \omega(\log n)$.

Filtering Bounded Knapsack Constraints

In this paper, we consider the generalization of the knapsack problem where there may exist multiple copies of each item. This setting is actually much more common as in CP the domains of variables are rarely binary.

Formal Background

Definition 3 (Bounded Knapsack Problem (BKP)). *Given n items $X = \{x_1, \ldots, x_n\}$, profits $p_1, \ldots, p_n \in \mathbb{N}$, weights $w_1, \ldots, w_n \in \mathbb{N}$, numbers of copies of each item $u_1, \ldots, u_n$, and a capacity $C \in \mathbb{N}$, the* bounded knapsack problem (BKP) *consists in finding an assignment $X_i \leftarrow x_i \in \{0, \ldots, u_i\}$ for all i such that $\sum w_i x_i \leq C$ and the total profit $\sum p_i x_i$ is maximized.*

Analogously to the binary case, we can compute an upper bound on this maximization problem: After arranging the items in non-increasing order of efficiency, we greedily include all copies of the most efficient items, until doing so would exceed the capacity of the knapsack, i.e., until we have reached the item x_s such that $\sum_{j=1}^{s-1} w_j u_j \leq C$ and $\sum_{j=1}^{s} w_j u_j > C$. We say again that x_s is the *critical item* for our knapsack instance. We then select the maximum number of copies plus some fraction of another copy of x_s that can fit into the knapsack, i.e., $C - \sum_{j=1}^{s-1} w_j u_j$ weight units of x_s (the integral part of this quantity divided by w_s is the number of full copies of x_s plus some fractional part of one additional copy). The total profit is then $\tilde{P} = \sum_{j=1}^{s-1} p_j u_j + \frac{p_s}{w_s}(C - \sum_{j=1}^{s-1} w_j u_j)$.

item	1	2	3	4	threshold
profit	3	4	5	1	$B = 25$
weight	1	2	5	2	$C = 14$
copies	4	3	1	2	
efficiency	3	2	1	0.5	$\tilde{P} = 28$

Table 2: Bounded Knapsack Constraint.

Definition 4 (Bounded Knapsack Constraint). *Given a lower bound $B \in \mathbb{N}$, the (global) bounded knapsack constraint enforces that $\sum_i w_i X_i \leq C$ and $\sum_i p_i X_i \geq B$, and for the domain of each X_i it holds $D_i \subseteq \{0, \ldots, u_i\}$.*

Note that we do not need to consider lower bounds on the number of copies that must be included in the knapsack as these can obviously be set to zero after modifying accordingly the upper bounds, the profit threshold B, and the capacity C. Furthermore note that, if $\tilde{P}_{[X_i \geq l_i]} \geq B$ and $\tilde{P}_{[X_i \leq u_i]} \geq B$, then $\tilde{P}_{[X_i = b_i]} \geq B$ for all $b_i \in \{l_i, \ldots, u_i\}$. Consequently, we will focus on computing new upper and lower bounds for each variable domain. As computing the upper bounds works analogously to the algorithm that we present in the following, we will focus here on the computation of new lower bounds for each variable domain.

The motivation for considering bounded instead of binary knapsacks is the following. Of course, we could model a bounded knapsack problem by replicating multiple copies of the same item. While this is neither elegant nor efficient, the real problem is lacking filtering effectiveness. Assume that the binary representation of a bounded knapsack constraint infers that one copy of an item is mandatory. Then, for reasons of symmetry, this also holds for all other copies! This is bad news, as it implies that the binary representation will *only* ever filter an item when its lower bound can be set to its upper bound! Consequently, filtering will be largely ineffective and search trees will be large. The bounded knapsack constraint, on the other hand, allows to infer arbitrary upper and lower bounds on the variables' domains.

Filtering BKPs in Amortized Linear Time

Given a bounded knapsack constraint, we can easily compute the critical item in linear time once the efficiency ordering is known. Analogous to the binary case, beginning at the critical item we can compute the values $\tilde{w}_i$ in one linear scan (Sellmann 2003).

Example: Consider the bounded knapsack constraint in Table 2. The linear continuous relaxation inserts four copies of item x_1, three copies of x_2, and 80% of x_3, which is the critical item. The value of the relaxation is $\tilde{P} =$

$4 * 3 + 3 * 4 + 0.8 * 5 = 28$, which is greater than the profit threshold $B = 25$.

To determine a new lower bound on the number of copies of x_1, we compute $\tilde{w}_1$. We can afford to replace 1 weight unit of x_1 with those remaining 20% of the critical item as $1(e_1 - e_3) = 2 \leq 3 = \tilde{P} - B$. However, we cannot afford to replace additionally all weight units of all copies of x_4 as $1(e_1 - e_3) + 4(e_1 - e_4) = 2 + 10 > 3 = \tilde{P} - B$.

We have now found that the new position is x_4. We compute $\tilde{w}_1$, which we already know is between 1 and 5: $\tilde{w}_1 = \frac{7}{5}$ since $1(e_1 - e_3) + 0.4(e_1 - e_4) = 2 + 1 = 3 = \tilde{P} - B$. Consequently, we cannot afford to lose more than 1.4 weight units of x_1, one copy of which weighs 1. And thus, we can afford to lose at most $\lfloor \frac{1.4}{1} \rfloor = 1$ copy of x_1. Therefore, the number of copies of x_1 which are mandatory is $u_1 - \lfloor \frac{1.4}{1} \rfloor = 3$.

For x_2, we have that $\tilde{w}_2 \geq \tilde{w}_1 = \frac{7}{5}$. We begin the search for the new position at the old and find that replacing all weight units of all copies of x_4 is infeasible: $1(e_2 - e_3) + 4(e_2 - e_4) = 1 + 6 = 7 > 3 = \tilde{P} - B$. We find $\tilde{w}_2 = \frac{7}{3}$ as $1(e_2 - e_3) + \frac{4}{3}(e_2 - e_4) = 1 + 2 = 3 = \tilde{P} - B$. As the weight of one copy of x_2 is 2, we can afford to lose at most $\lfloor \frac{7}{3*2} \rfloor = 1$ copy. Therefore, the number of copies of x_2 which are mandatory is $u_2 - 1 = 2$.

Note again that the current position of the last fractional item used in the computation of $Q(\tilde{w}_i)$ increases monotonically with $\tilde{w}_i$, and consequently the total effort for these computations is linear once the efficiency order is known.

Filtering BKPs in Expected Sublinear Time

We propose to handle bounded knapsack constraint directly, without reverting to a binary representation, and therefore achieve much better filtering effectiveness. Our challenge is to compute the critical item and the number of mandatory copies faster than by a linear scan. Following the same approach as in (Katriel et al. 2007), to compute the critical item and the quantities $\tilde{w}_i$ we use finger trees as our core data structures (Mehlhorn 1980). Finger trees are essentially binary search trees with one shortcut from each node to its right brother (or first cousin, or second cousin, etc) on the same level. These trees allow us to quickly decide which items i need to be analyzed and also to compute the corresponding quantities $\tilde{w}_i$, whereby the cost for these computations is logarithmic in the number of items that we can skip over. Consequently, we can hope for a sublinear effort when the number of items whose bounds may change (we call these the analyzed items) is small, while the finger tree data structure still ensures that the total effort for this task will be linear in the worst case.

In particular, we maintain three finger trees. For the first two, at the leaf level we store the weight or profit *times the number of copies of each item*, whereby the order of the items belonging to leaves from left to right is given by decreasing efficiency of the items. Internal nodes in the tree store the *cumulative* weight or cumulative profit of all leaves in the corresponding subtree. In the final finger tree, we store again the weight times the number of copies of each item at the leaf level, in the same ordering of leaf nodes as before. However, in this tree internal nodes store the *maximum* weight of any leaf in the corresponding subtree.

We use the first two finger trees to compute the critical item and the quantities $\tilde{w}_i$, and the third finger tree for finding the next potentially mandatory item as in (Katriel et al. 2007). The two latter operations, finding the next candidate item and computing its corresponding value $\tilde{w}_i$, take time logarithmic in the number of items *between the position of the old and the new leaf*. If we need to consider k items for filtering, the total time is thus bounded by $O(k \log(n/k))$. For large k close to n, this gives a worst-case linear time complexity. As we mentioned in our review of the binary case, k is bounded by the length of the longest increasing sub-sequence of weights when items are ordered with respect to efficiencies. When we assume that the *total weight* of each item is independent of its efficiency, we know that the length of the longest increasing subsequence of total weights is in $k \in O(\sqrt{n})$ (Aldous & Diaconis 1999). Consequently, for the filtering task we achieve an expected runtime in $O(\sqrt{n} \log n)$. That is to say, in the bounded case it is even *more likely* that skipping will occur often as long as the per copy weights and/or the numbers of copies per item are independent of the items' efficiencies.

So far the adaptation of the algorithm in (Katriel et al. 2007) was straight forward. The actual challenge for us is to efficiently update the data-structures that the filtering is based on. Such an update is necessary right before filtering whenever the domain bounds have changed (by our own or by other constraints). Note that fast incremental updates are key to practical efficiency. In our experiments, two thirds of the propagation-time is spent on updates.

In the binary case, we can simply update the finger-trees by walking from the affected leaves to the root for each item that has changed its status. Since the height of the trees is logarithmic and since each item can change its status at most once on each path down the search tree, the total update effort for the entire path is bounded by $O(n \log n)$, and thus amortized at most linear when the search tree is at lest $\Omega(\log n)$ deep. In BKPs, however, the bounds on the number of copies can change multiple times for each item, and consequently the update method from (Katriel et al. 2007) can require $\Theta(cn \log n)$ (where c is the number of copies per item) for all updates on a path down the tree. Consequently, for large c the existing method has amortized worst-case super-linear runtime when used for BKPs.

We propose the following procedure: When the number of mandatory copies of an item is increased above zero then we subtract the corresponding amount from the capacity, the profit threshold, and the number of available copies of this item. In this way, the internal lower bound for each item remains zero. To adjust the upper bound, we only need to update the number of available copies. In both cases, we need to adjust our three finger trees as the total weight and profit of items have changed. The important idea is to perform these updates for all items that have changed their bounds *simultaneously*. We update the finger trees level by level, from the leaf level up, whereby on each level we only consider those nodes whose value has changed. We thus prevent

# Items		100			1,000			10,000	
Algorithm	bin	lin	bkp	bin	lin	bkp	bin	lin	bkp
UC 1%	23	42	24	99	352	27	1.0K	3.3K	52
UC 2%	14	39	17	96	332	16	866	3.0K	71
UC 5%	10	35	10	91	324	10	937	3.2K	40
UC 10%	9.8	32	6.4	86	315	8.2	854	3.1K	39
WC 1%	15	37	11	98	343	13	1.0K	3.4K	43
WC 2%	12	35	7.3	96	344	10	990	3.4K	40
WC 5%	11	35	5.0	90	341	7.7	937	3.4K	40
WC 10%	9.5	34	4.0	81	333	6.6	875	3.3K	39
SC 1%	14	36	8.7	98	336	12	1.0K	3.3K	44
SC 2%	11	35	5.7	97	332	9.4	993	3.3K	40
SC 5%	10	34	4.0	91	326	6.8	938	3.2K	39
SC 10%	9.8	32	3.3	86	317	6.6	871	3.1K	38
AS 1%	15	36	8.6	97	336	12	1.0K	3.3K	44
AS 2%	12	34	5.7	95	333	8.9	1.0K	3.3K	40
AS 5%	8.7	33	3.6	86	326	6.9	957	3.2K	39
AS 10%	8.3	32	3.0	70	318	5.9	890	3.1K	38

Table 3: Knapsack filtering in the presence of a profit threshold of 1%, 2%, 5%, or 10% below the linear relaxation value. We report the average CPU-time [μ-sec] of the sublinear time filtering algorithm by (Katriel et al.'07) on the binary representation (*bin*), the linear-time bounded knapsack filtering algorithm by (Sellmann'03) (*lin*), and the algorithm introduced in this paper (*bkp*) on benchmark sets with 100 instances each. Instances have 100, 1,000, or 10,000 items each and have uncorrelated (UC), weakly correlated (WC), strongly correlated (SC), or almost strongly correlated (AS) profits and weights. The number of copies of each item was drawn uniformly at random in $\{1, 2, 5, 10, 20, 50, 100\}$.

that nodes on higher levels in the tree get updated multiple times and we are therefore sure to touch each node in the finger tree at most once. This already gives us a worst-case linear time guarantee. A more careful analysis shows that, when d items change their bounds, we will touch at most $O(d(\log(\frac{n}{d}) + 1))$ nodes in the tree. It follows:

Theorem 1. *For a previously initialized bounded knapsack constraint with n items, assume that the capacity, profit threshold, and lower and upper bounds of d items have changed. Assume further that our filtering algorithm can set tighter bounds on l items while analyzing another k items whose bounds do not change. The total effort is then bounded by $O(d \log(\frac{n}{d}) + l \log(\frac{n}{l}) + k \log(\frac{n}{k}))$. When the total weight of items is independent of the items' efficiency, we expect $k \in O(\sqrt{n})$. Therefore, when the number of items that change their bounds is not extremely large (for example, when $d, l \in O(\frac{n}{\log n})$), we expect a sublinear running time. In the worst-case, filtering takes at most linear time.*

Numerical Results

Theorem 1 is very encouraging, but it is based on the assumptions that the longest increasing subsequences are not too long, and that not too many items change their bounds between two invocations of the filtering algorithm. If this was not the case, we would be better off using the worst-case linear-time algorithm from (Sellmann 2003) which requires much less maintenance of internal datastructures. Therefore, in this section we test and quantify the speed-ups that the new method achieves in practice.

# Items	1,000			10,000		
Algorithm	bin	lin	bkp	bin	lin	bkp
copies 1	14	332	14	39	3.3K	39
2	14	330	14	74	3.3K	39
5	21	330	14	184	3.3K	40
10	37	331	14	373	3.3K	40
20	71	331	14	745	3.3K	40
50	175	330	14	1.8K	3.3K	40
100	357	331	13	3.5K	3.3K	40

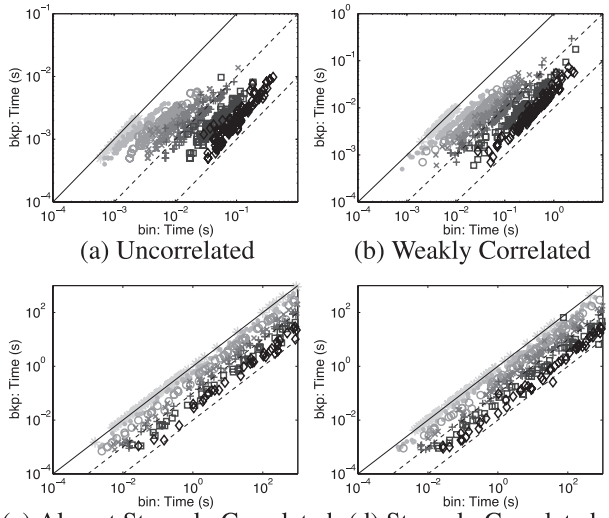
Table 4: Knapsack filtering for a profit threshold of 2% below the linear relaxation value. We report the average CPU-time [μ-sec] of the sublinear-time filtering algorithm by (Katriel et al.'07) on the binary representation (*bin*), the linear-time bounded knapsack filtering algorithm by (Sellmann'03) (*lin*), and the algorithm introduced in this paper (*bkp*) on benchmark sets with 100 instances each. Each instance has either 1,000 or 10,000 items with uncorrelated profits and weights. In each benchmark set we keep the number of copies per item the same and vary this number between 1, 2, 5, 10, 20, 50, and 100 copies of each item.

We first conduct the analogous experiment as in (Katriel et al. 2007): We filter bounded knapsack constraints which are generated according to the established standard knapsack distributions where profits and weights of items are either uncorrelated, weakly correlated, almost strongly correlated, or strongly correlated (Pisinger 2005; Katriel et al. 2007). For each item we randomly select a maximum number of copies in $\{1, 2, 5, 10, 20, 50, 100\}$. The profit threshold is set to 1%, 2%, 5%, or 10% below the linear relaxation value. All experiments were run on a Dell 1855 blade with two Xeon 800MHz processors and 8GB of RAM.

Table 3 summarizes the runtime comparison between the binary representation where we introduce one binary variable for each copy of each item (*bin*), the linear-time filtering algorithm that achieves relaxed consistency for bounded knapsacks by (Sellmann 2003) (*lin*), and the approach presented in this paper (*bkp*). We observe that the old linear-time state-of-the-art for bounded knapsacks by (Sellmann 2003) is not competitive. Algorithm *bkp* is up to 80 times faster than algorithm *lin*, whereby the speed-ups scale up with the size of the instances.

Note that *lin* and *bkp* obtain the exact same filtering power. Algorithm *bin*, however, is much less effective in its filtering as it can only infer that *all* copies of an item are either all mandatory or all forbidden. Therefore, we would have expected that *bin* runs faster at the cost of being less effective. However, this is not the case. The reason why it takes up to 25 times more time than *bkp* is that it needs to perform a lot more work when indeed all copies of an item are consecutively found to be mandatory or forbidden.

In Table 4 we present the filtering times for uncorrelated knapsack instances where all items have 1, 2, 5, 10, 20, 50, or 100 copies each. We see clearly that the increase in the number of copies does not affect the algorithms *lin* and *bkp* which handle bounded knapsacks directly, while the binary representation becomes more and more inefficient as the maximum number of copies per item increases. This is again due to the fact that *bin* needs to consider each individ-



(a) Uncorrelated (b) Weakly Correlated

(c) Almost Strongly Correlated (d) Strongly Correlated

Figure 1: Optimizing 100 item BKP instances with algorithms *bin* and *bkp*. Instances with 1 copy per item are light stars, 2 copies light dots, 5 grey circles, 10 grey Xs, 20 dark grey crosses, 50 dark grey squares, 100 black diamonds.

ual copy of an item when filtering occurs.

The real advantage of using *bkp* over *bin* is however when an actual tree-search is conducted. Figure 1 compares the times needed to optimize bounded knapsack problems sampled from the standard instance distributions using *bin* and *bkp*. For *bkp*, we branch on the critical item and round it down first (and up upon backtracking). For *bin* we emulate the same branching behavior by setting the appropriate number of variables belonging to copies of this item to zero (or one upon backtracking). We observe up to two orders of magnitude speed-ups of *bkp* over *bin* mostly due to the fact that *bkp* visits much fewer search nodes thanks to its better filtering effectiveness.

The latter is amplified when we consider problems with more than one bounded knapsack constraint. In Figure 2, we show the times and number of search nodes when optimizing instances with 50 items and three uncorrelated bounded knapsack constraints. For both approaches, we branch on one of the critical items whereby we pick the one where most of the other BKP constraints agree whether it should be rounded down or up (i.e., depending on whether it is more or less efficient than the critical item of that constraint). We first branch in the direction of that bias introduced by the other constraints. We observe that the improved filtering power of *bkp* reduces the number of nodes by up to three orders of magnitude (some outliers are caused by diverging branching behavior caused by a lack of filtering in *bin*). Analyzing our data, we found that *bkp* visits on average 99.5% less search nodes that *bin* needs to consider. On average, *bkp* then works more than 750 times faster.

Conclusion

We introduced a fast filtering algorithm for bounded knapsack constraints which is based on the skipping heuristic introduced in (Katriel et al. 2007). We proved that this filtering algorithm runs in expected sublinear time when the number of items that change their bounds is not excessively large and when the total weights of all copies of the items are independent of their efficiencies.

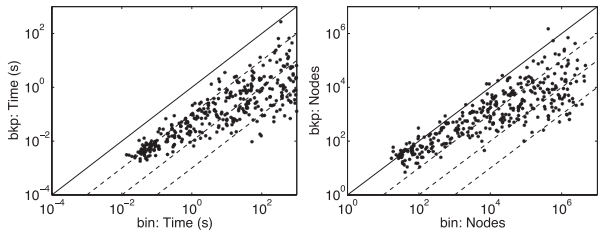


Figure 2: Maximizing instances with 50 items and randomly selected numbers of copies in $\{1, 2, 5, 10, 20, 50, 100\}$ with algorithms *bin* and *bkp*. Each instance consists of three BKP constraints with weights uncorrelated to profits.

Numerical results show that the new method massively outperforms the linear-time state-of-the-art relaxed consistency algorithm for bounded knapsack constraints by (Sellmann 2003). For large numbers of items, the new method is almost up to two orders of magnitude faster while achieving the same filtering power.

We also compared the new method against a sublinear filtering algorithm for a binary representation of the bounded knapsack constraint. The new algorithm has strictly better filtering effectiveness, it never works slower and often faster per search node depending on the number of copies per item. Systematic search experiments on problems with one or more bounded knapsack constraints show that handling bounded knapsack constraints directly is highly desirable as the stronger filtering power frequently reduces the number of choice points by several orders of magnitude.

Acknowledgement: We would like to thank Wadeck Follonier for his help in implementing the initial version of bounded knapsack constraint in the constraint solver JaCoP.

References

Aldous, D., and Diaconis, P. 1999. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bull. of the Amer. Math. Society*, 36(4):413–432.

Brodal, G. S. 2005. Finger search trees. In *Handbook of Data Structures and Applications*. CRC Press.

Dantzig, G. 1957. Discrete variable extremum problems. *Operations Research*, 5:226–277.

Fahle, T., and Sellmann, M. 2002. Cost-based filtering for the constrained knapsack problem. *AOR*, 115:73–93.

Fahle, T., and Sellmann, M. 2000. Constraint Programming Based Column Generation with Knapsack Subproblems. *CPAIOR*, 33–43.

Katriel, I., Sellmann, M., Upfal, E., Van Hentenryck, P. 2007. Propagating Knapsack Constraints in Sublinear Time. *AAAI*, 231–236.

Mehlhorn, K. 1980. A New Data Structure for Representing Sorted Lists. *WG*, 90–112.

Pisinger, D. 2005. Where are the hard knapsack problems? *Computers and Operations Research*, 32:2271–2282.

Sellmann, M. 2003. Approximated consistency for knapsack constraints. *CP'03*, 679–693.