Propagating Knapsack Constraints in Sublinear Time*

Irit Katriel, Meinolf Sellmann, Eli Upfal, and Pascal Van Hentenryck
Brown University, PO Box 1910, Providence, RI 02912
{irit,sello,eli,pvh}@cs.brown.edu

Abstract

We develop an efficient incremental version of an existing cost-based filtering algorithm for the knapsack constraint. On a universe of n elements, m invocations of the algorithm require a total of $O(n \log n + mk \log(n/k))$ time, where $k \leq n$ depends on the specific knapsack instance. We show that the expected value of k is significantly smaller than n on several interesting input distributions, hence while keeping the same worst-case complexity, on expectation the new algorithm is faster than the previously best method which runs in amortized linear time. After a theoretical study, we introduce heuristic enhancements and demonstrate the new algorithm's performance experimentally.

Introduction

Constraint programming (CP) exploits a combination of inference and search to tackle difficult combinatorial optimization problems. Inference in CP amounts to filtering the variable domains, often by dedicated algorithms exploiting the underlying structure in constraints. Search consists of dividing the resulting problem in two or more parts (e.g., by splitting a variable domain) giving new opportunities for inference on the subproblems. An important issue in CP is to strike a good balance between the time spent on filtering and the induced reduction in the search space.

Traditionally, filtering algorithms are analyzed using worst-case complexity and in isolation. In practice, however, filtering takes place inside a propagation algorithm which is rarely adversarial. Moreover, a CP engine repeatedly applies filtering to a sequence of only moderately changing problems so that incremental behavior is a significant aspect. Thus, it is best to analyze filtering algorithms incrementally (e.g. by considering the cost along a branch of the search tree) using *expected-case* complexity analysis. These considerations led to the design of a filtering algorithm with an improved *expected* complexity for *alldifferent* constraints (Katriel & Van Hentenryck 2006).

This paper pursues a similar approach for cost-based filtering of knapsack constraints. It shows how to improve the best previous algorithm by (Fahle & Sellmann 2002) both theoretically (in the expected sense) and experimentally. The new algorithm is based on a fundamental dominance property arising in knapsack constraints, which can be used to significantly reduce the number of items that need to be considered during filtering. The resulting algorithm is shown to run in expected sublinear time under reasonable probabilistic assumptions, while matching the complexity of the previous best method in the worst case. Moreover, experimental results using Pisinger's knapsack benchmarks indicate that a practical implementation of the new algorithm may reduce filtering time by two orders of magnitude. As such, these results shed new light on the benefits of designing and analyzing filtering algorithms for global constraints beyond worst-case complexity and with incremental and expected-case complexity models in mind, importing new tools from theoretical computer science into CP.

This paper first reviews the filtering algorithm in (Fahle & Sellmann 2002), providing the necessary intuition for the rest of the paper. The new algorithm is then presented and analyzed theoretically both in the worst and in the expected case. Experimental results on a wide variety of knapsack instances demonstrate the effectiveness of the new algorithm.

Cost-Based Filtering for Knapsack

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 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 1 iff $x_i \in S$. In the linear relaxation of the knapsack problem, 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 select 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 $\sum_{j=1}^{s-1} w_j \leq C$ and $\sum_{j=1}^{s} w_j > C$. 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 is then $\tilde{P} = \sum_{j=1}^{s-1} p_j + \frac{p_s}{w_s}(C - \sum_{j=1}^{s-1} w_j)$.

*This work was supported in part by the National Science Foundation through the Career: Cornflower Project (award number 0644113), and by NSF awards CCR-0121154 and DMI-0600384, as well as ONR Award N000140610607.
Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Given a lower bound $B \in \mathbb{N}$, the (global) knapsack constraint enforces that $\sum_i w_i X_i \leq C$ and $\sum_i p_i X_i \geq B$. Achieving GAC 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 cannot belong (in full) to a fractional solution whose profit is at least B, then it also cannot belong to an integral solution of this quality. 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., without x_i we cannot achieve a relaxed profit of at least B. Then, we remove the value 0 from the domain of X_i. On the other hand, we classify x_i as *forbidden* iff the fractional optimum of the instance $(X \setminus \{x_i\}, C - w_i)$ has profit 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 the domain of X_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.

Knapsack Filtering in Amortized Linear Time

In (Fahle & Sellmann 2002), an algorithm was developed that can identify the mandatory and forbidden 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 weight. Thus, when the status of a number of items (to be included in or excluded from the knapsack) or the current bound changes, the remaining items are already sorted and the computation can be re-applied in linear time. The total time spent on m repetitions of the algorithm on the same constraint is therefore $\Theta(n \log n + mn)$. We describe this algorithm below.

For each $i < s$, let x_{s_i} denote the critical item of the instance $(X \setminus \{x_i\}, C)$. Clearly, $s_i \geq s$. The maximum profit of a solution that excludes x_i is

$$\tilde{P}_{\neg i} = \sum_{j=1}^{s_i-1} p_j + \frac{p_{s_i}}{w_{s_i}} \left(C - \left(\sum_{j=1}^{s_i-1} w_j - w_i \right) \right) - p_i.$$

If $\tilde{P}_{\neg i} < B$, then x_i is mandatory.

The core observation in (Fahle & Sellmann 2002) is the following: If, for two items x_i and x_j, we have that $w_i \leq w_j$, then $s_i \leq s_j$. Hence, if we traverse the items of $\{x_1, \ldots, x_s\}$ by non-decreasing weight, then all s_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 search for the next critical item can begin at the location of the current critical item, and it always advances in one direction. If we constantly keep track of the sum of weights and the sum of profits of all items up to the current critical item, we only need to spend linear time to determine all mandatory items.

Symmetrically, for $i > s$, let x_{s_i} be the critical item of the instance $(X \setminus \{x_i\}, C - w_i)$. Then $s_i \leq s$ and the maximum profit of a solution that includes x_i is

$$\tilde{P}_i = \sum_{j=1}^{s_i-1} p_j + \frac{p_{s_i}}{w_{s_i}} \left(C - \left(\sum_{j=1}^{s_i-1} w_j + w_i \right) \right) + p_i.$$

If $\tilde{P}_i < B$, then x_i is forbidden.

Again, if we traverse the items in $\{x_s, \ldots, x_n\}$ by non-decreasing order of weight, each critical item is always to the left of the previous critical item, and all computations can be performed with a single linear scan.

Filtering by the Efficiency Ordering

The same time bound can be obtained when considering the items by non-increasing order of efficiency (Sellmann 2003): We first identify the critical item x_s as before. Then, instead of computing the critical item of the instance $(X \setminus \{x_i\}, C)$, for each x_i, we compute the portion of each x_i that we can give up and still satisfy the constraint.

More precisely, 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 from the items that are not part of the fractional knapsack solution. Let $Q(w)$ be the profit from the best w units of weight among those items, i.e., the $C - (\sum_{j=1}^{s_i-1} w_j - w_i)$ unused units of x_s and the items $\{x_{s+1}, \ldots, x_n\}$. Then $\tilde{w}_i$ is the largest value such that $\tilde{P} - \tilde{w}_i e_i + Q(\tilde{w}_i) \geq B$.

If $\tilde{w}_i \geq w_i$, this means that we can give up all of x_i, i.e., it is not mandatory. Note that for i, j such that $e_i \geq e_j$, $\tilde{w}_i \leq \tilde{w}_j$. Therefore, traversing the items by non-increasing efficiency allows us to identify all mandatory items in a single linear scan as before. A symmetric computation can determine which items are forbidden. Using this approach, we do not need to sort the items twice (once by efficiency and once by weight). The solution we describe in the next section is based on this algorithm.

Filtering in Sublinear Expected Time

In this section we describe our alternative filtering algorithm. It requires $O(n \log n)$-time preprocessing (to sort the items by efficiency) and every repetition of the propagator requires $O(d \log n + k \log(n/k))$ additional time, where d is the number of elements whose status was determined (potentially by other constraints) to be mandatory or forbidden in the last branching step, and $k \leq n$ is a parameter determined by the knapsack instance. Since the status of an element can be settled at most once, the first term of the complexity sums to $O(n \log n)$ throughout the whole sequence and we get that, in total, m repetitions of the propagator take $O(n \log n + mk \log(n/k))$ time. Note that $k \log(n/k) = O(n)$, so our algorithm is never slower than the previous one. In the next section, we will show that the expected value of k is significantly smaller than n for several interesting input distributions.

Lemma 1. *Let $x_i, x_j \in \{x_1, \ldots, x_s\}$ 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.*

Proof. As the items are arranged by order of non-increasing efficiency, the average efficiency decreases as an interval of items grows to include higher-indexed items. Consequently, $w_j \leq w_i$ implies $Q(w_j)/w_j \geq Q(w_i)/w_i$, and therefore, $w_i(e_i - Q(w_i)/w_i) \geq w_j(e_j - Q(w_j)/w_j)$. We also know that x_i is not mandatory, so $\tilde{P} - w_i e_i + Q(w_i) \geq B$. Then, $\tilde{P} - w_j e_j + Q(w_j) = \tilde{P} - w_j(e_j - Q(w_j)/w_j) \geq \tilde{P} - w_i(e_i - Q(w_i)/w_i) \geq B$, that is, x_j is not mandatory. □

We focus our attention on the task of identifying the mandatory items. The forbidden items can then be found by a symmetric algorithm. The algorithm that we propose traverses the items of $\{x_1, \ldots, x_s\}$ by *non-increasing* efficiency (breaking ties in favor of larger weight). At all times, we remember the maximum weight $w_{\max}$ of an item that was checked and was *not* found to be mandatory. According to Lemma 1, an item x_i only needs to be checked if $w_i > w_{\max}$ when x_i is considered. As before, to check an item x_i, we compute $\tilde{w}_i$. If this value is less than w_i, we conclude that x_i is mandatory. At this stage we do not do anything about it, except to insert x_i to a list of items that were found to be mandatory. On the other hand, if $\tilde{w}_i$ is larger than w_i, we set $w_{\max} \leftarrow \max\{w_{\max}, w_i\}$.

We next describe two data structures that allow us to perform this task efficiently: the item-finding search tree $\mathcal{T}_f$ which helps us to quickly find the next item that needs to be checked; and the computation tree $\mathcal{T}_c$ which speeds up the $\tilde{w}_i$ computations. Finally, we put all the pieces together and describe the complete algorithm.

The Item Finding Search Tree: We place the items of $\{x_1, \ldots, x_s\}$ at the leaves of a binary tree $\mathcal{T}_f$, sorted from left to right by non-increasing efficiency (breaking ties in favor of higher weight). Every internal node contains the maximum weight of an item in its subtree.

In each application of the propagator, we begin traversing the nodes at the leftmost leaf of the tree, i.e., the most efficient item. As described above, we always remember the maximum weight $w_{\max}$ of an item that was examined and was not determined to be mandatory. Once we have finished handling an item x_i, we use the tree to find the next item to be examined, $next(x_i)$. This item is represented by the leftmost leaf whose weight is higher than $w_{\max}$, among all leaves that are to the right of x_i.

One way to perform this search is to climb from x_i towards the root; whenever the path to the root goes from a left-son to its parent, we look at the right-son of this parent. If there is a leaf in the right-son's subtree whose weight is higher than $w_{\max}$, the search proceeds in this subtree. Otherwise, we continue climbing towards the root. With this method, finding the next item takes $O(\log n)$ time. Using standard finger-tree techniques (e.g., (Brodal 2005)), we reduce this further: For every level of the tree, we create a linked list that connects the tree nodes of this level from left to right. These links can be used as shortcuts in the search, and as a result the search time is reduced from $O(\log n)$ to $O(\log \ell_i)$, where ℓ_i is the number of items between x_i and $next(x_i)$. Looking up k items in sorted order then takes $\sum_{j=1}^{k} O(\log \ell_j)$. Since $\sum_{j=1}^{k} \ell_j \leq n$, the total time spent is $O(k \log(n/k))$.

Computing the $\tilde{w}_i$ Values: Our second data structure is a binary tree $\mathcal{T}_c$ which we use to compute the $\tilde{w}_i$ values. The leaves of $\mathcal{T}_c$ hold all items $\{x_1, \ldots, x_n\}$, sorted in the same order as the leaves of $\mathcal{T}_f$. Internal nodes hold the sums of the weights and profits of the items in their subtree.

Given the knapsack capacity C, we can compute the critical item x_s, as well as the total profit of the optimal fractional solution, in $O(\log n)$ time as follows. Our search algorithm receives the pair (r, C) where r is the root of the tree and C is the capacity of the knapsack. Let $Left(r)$ and $Right(r)$ be the left and right children of r and let $W_{Left(r)}$ be the total weight of the items in the subtree rooted at $Left(r)$. If $W_{Left(r)} > C$, then the search proceeds to the left subtree. Otherwise, we recursively search using the parameters $(Right(r), C - W_{Left(r)})$. That is, we search for the critical item in the right subtree, but we take into account the weight of the left subtree. Once the search reaches a leaf, it has found the critical item. During the search, we also keep track of the total profit up to the critical item.

To compute the value $\tilde{w}_i$, we search in a similar manner starting from the critical item x_s, and continue to the right, skipping subtrees that do not contain the solution. The way to do this is to compute the first location in which the *loss* exceeds the slack $\tilde{P} - B$ of the current solution, i.e., the first index i for which $\tilde{w}_i(e_i - Q(\tilde{w}_i)/\tilde{w}_i) \geq \tilde{P} - B$. Since the items are sorted by efficiency, we know that, if the loss of a certain subtree t is not too large, then the loss of any proper subtree of t is also not too large, so t can be skipped altogether. In other words, we can apply a finger-tree search again and find all $\tilde{w}_i$'s within $O(k \log(n/k))$ time.

Putting the Pieces Together: We assume that the propagator is called m times, each time on a reduced instance, i.e., an instance which is similar to the previous one, except that some items may be missing because their status has been determined, the knapsack capacity may have decreased, and the profit target may have increased. Since these changes may affect the position of the critical item x_s, we construct $\mathcal{T}_f$ over the complete set of items, and not only over $\{x_1, \ldots, x_s\}$. When searching for the next item to examine, we do not look beyond x_s.

The trees $\mathcal{T}_f$ and $\mathcal{T}_c$ are constructed once, at a cost of $O(n \log n)$ time for sorting. As a simplification, i.e., to avoid the need for complex data structures, we may assume that the trees are static in that they contain a leaf for every item and an item is deleted by marking its leaf as *empty*. An *empty* leaf, of course, does not contribute to the contents of the internal nodes. Inserting or deleting an item then requires only to update the internal nodes along a single root-leaf path, which takes $O(\log n)$ time.

Consider a single call to the propagator, and assume that, while it is executed, k items were checked and ℓ were found to be mandatory. The total time for identifying and checking the items, as described above, is $O(k \log(n/k))$. After all necessary checks have been performed, we have a list of items that became mandatory. We can exclude them from further consideration after updating the knapsack capacity and the profit bound to reflect the fact that they are in the knapsack in every solution. In other words, let M be the set of items that were determined to be mandatory. Let W be their total weight and P their total profit. Our knapsack instance can be reduced to the instance $(X \setminus M, C - W)$, with the profit goal $B - P$. It remains to update the two trees with respect to these changes. As mentioned before, each item deletion takes $O(\log n)$ time to update the data in the internal nodes along the relevant root-leaf path. We have $|M|$ such deletions, for a total of $O(|M| \log n)$. Similarly, we have a list F of items that were determined to be forbid-

den. We remove them from the trees in $O(|F| \log n)$ time. Since each item can be deleted at most once, this adds up to $O(n \log n)$ time for all m repetitions of the propagator. Note that the previous critical item x_s remains the critical item after these changes.

Finally, we need to be able to update the data structures upon external changes to the variable domains. If items are specified as mandatory or forbidden, we handle this in the same way as described earlier (it does not matter that this is an external change and not a decision made by our algorithm). When C becomes smaller, we need to find a new critical item. Since the new critical item can only be to the left of the old one, we can find it by a finger search, which adds up to $O(n)$ for all searches. When B changes externally, this does not require updating the search trees directly; the updates will occur when the propagator is re-applied.

Complexity Analysis

Worst Case: We have shown that the total time spent is $O(n \log n)$ in addition to $O(k_i \log(n/k_i))$ for the i-th invocation of the propagator, where k_i is the number of items that were tested and whose status was not settled. In the worst case, $k_i = n$ and then the amortized time spent is $O(n)$, as in (Fahle & Sellmann 2002). Typically, however, there will be dominated items that we can skip over. Our purpose in this section is to analyze the expected value of k_i.

We determine an upper bound on the the expected value of k_i for various distributions of knapsack instances. By Lemma 1, we know that the weights of the k_i items are increasing. Therefore, an upper bound on k_i is the expected length of a *longest increasing subsequence* (LIS) of $w_1, w_2, \ldots, w_n$. In the symmetric computation, which identifies the forbidden items, k_i is the length of the longest *decreasing* subsequence (LDS).

We analyze the expected length of an LIS and an LDS with respect to four different distributions which are widely considered in the knapsack literature: In *uncorrelated instances*, profits and weights are drawn independently. *Weakly correlated instances* are generated by choosing profits uniformly in a limited interval around the corresponding weights. Instances are *strongly correlated* when the profit of every item equals its weight plus some constant. In *subset sum instances*, profits and weights are identical.

Subset Sum: Here, the LISs and LDSs have lengths 1 when efficiency ties are broken in favor of higher (lower) weight for the mandatory (forbidden) items.

Strongly Correlated: Here, the efficiency ordering of items coincides with an ordering with respect to increasing weights. Consequently, we see the worst-case lengths for the LISs here. The LDSs, on the other hand, only have length 1.

Weakly Correlated: It is known that the expected length of an LIS (or LDS) of a random permutation of n items is $2\sqrt{n} - o(\sqrt{n})$ (Aldous & Diaconis 1999). This immediately gives that $Ek_i = O(\sqrt{n})$ if the input is created by selecting each weight and each efficiency independently from $U[0,1]$.

Uncorrelated: Here, sorting the items by non-increasing efficiency does not give a random permutation of the weights: The larger weights are more likely to appear towards the end of the list, so the LIS' are longer. We will show:

Theorem 1. *Let $w_1, w_2, \ldots, w_n$ and $p_1, p_2, \ldots, p_n$ be independently distributed $U[0,1]$, such that $w_1/p_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$. Then, with high probability, the length of the LIS of the sequence $w_1, w_2, \ldots, w_n$ is bounded by $O(n^{2/3})$.*

For a fixed sequence of weights and profits chosen at random we have:

Lemma 2. *Let $w_1 \leq w_2 \leq \cdots \leq w_k$ be an increasing sequence and let each of $p_1, \ldots, p_k$ be chosen independently from $U[0,1]$. Then the probability that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \frac{p_k}{w_k}$ is at most $\frac{1}{k!}(\frac{1}{w_1})^{k-1}$.*

Proof. For each $1 < i \leq k$, let $q_i = p_{i-1}\frac{w_i}{w_{i-1}}$. We compute the probability that for all $1 < i \leq k$ we have $p_i \leq q_i$:

$$\int_0^1 \int_0^{\min[1,q_2]} \int_0^{\min[1,q_3]} \cdots \int_0^{\min[1,q_k]} 1 \; \mathrm{d}p_k \ldots \mathrm{d}p_2 \mathrm{d}p_1$$

$$\leq \frac{1}{k!} \prod_{j=2}^{k} (\frac{w_j}{w_{j-1}})^{k-j+1} = \frac{1}{k!} \prod_{j=2}^{k} (\frac{w_j}{w_1}) \leq \frac{1}{k!} (\frac{1}{w_1})^{k-1}.$$

□

The above bound depends on w_1, the smallest weight in the sequence. To use this lemma we need to show that in any long sequence with weights drawn independently in $U[0,1]$, "many" weights will not be "very small":

Lemma 3. *Let $w_1, w_2, \ldots, w_n$ be independently distributed $U[0,1]$. With high probability, at least $n - e^2 k$ of them are larger than k/n.*

Proof. The probability that $e^2 k$ of the weights are at most k/n is $\binom{n}{e^2 k} \left(\frac{k}{n}\right)^{e^2 k} \leq \left(\frac{en}{e^2 k} \frac{k}{n}\right)^{e^2 k} = e^{-e^2 k}$. □

We can now prove an upper bound on the expected length of the LIS for the uncorrelated input.

Proof of Theorem 1: If there is an increasing subsequence σ of length $2s = 2e^2 n^{2/3}$, then by Lemma 3, with high probability there is a subsequence σ' of σ of length $s = e^2 n^{2/3}$ consisting solely of elements whose weights are at least $n^{-1/3}$. By Lemma 2, the probability that such an increasing subsequence σ' of length s exists in the complete sequence of n items, is at most

$$\binom{n}{s} \frac{1}{s!} \left(\frac{e^2 n}{s}\right)^{s-1} \leq \left(\frac{en}{s} \frac{e}{s} \frac{e^2 n}{s}\right)^{s} \leq e^{-n^{2/3}}. \quad \square$$

Consequently, on uncorrelated knapsack instances, we expect our algorithm to run in amortized time $O(n^{2/3} \log n)$.

Practical Evaluation

After our theoretical study, we now discuss the practical side of knapsack filtering with respect to linear relaxation bounds. First, we describe some heuristic enhancements and implementation details of our algorithm. Then, we present numerical results which clearly show the benefits of the new filtering technique introduced in this paper.

Implementation and Heuristic Enhancements

The core ingredient of our algorithm are binary finger-trees. To eliminate the need of introducing slow and memory-costly pointers to parents, both sons, and the relative to the right, we simply store the elements of the tree in one consecutive array with first index 1. Then, the parent of node $i > 1$ is $\lfloor i/2 \rfloor$, the left son is $2i$, the right son is $2i+1$, and, for all $i \neq 2^z, z \in \mathbb{N}$, the relative to the right is simply $i+1$.

We also introduce two practical enhancements: First, in some cases we can terminate the computation of $\tilde{w}_i$ early. Note that, what we really perform when traversing our finger-tree is essentially a binary search. That is, whenever we move up or dive to the right son, we have found a new lower bound $\tilde{w}_i^{(l)} \leq \tilde{w}_i$. And whenever we move to the right brother (or cousin etc) or dive to the left son, we have found a new upper bound $\tilde{w}_i^{(u)} \geq \tilde{w}_i$. Therefore, whenever $\tilde{w}_i \geq \tilde{w}_i^{(l)} \geq w_i$, we already know that item x_i is not mandatory. Analogously, whenever $\tilde{w}_i \leq \tilde{w}_i^{(u)} < w_i$, we already know that item x_i is mandatory. Furthermore, we can deduce that item x_i is mandatory whenever replacing the remaining capacity $w_i - \tilde{w}_i^{(l)}$ with the efficiency of the item that determines $\tilde{w}_i^{(l)}$ is not enough. In all these cases, we can terminate our computation early and continue with the next item right away.

Second, when an item x_i is found not to be mandatory, we can safely continue with the next item $x_j \in \{x_{i+1}, \ldots, x_s\}$ for which w_j is even greater than $\tilde{w}_i \geq w_i$, because we know that $\tilde{w}_j$ is always greater or equal $\tilde{w}_i$. We refer to this enhancement as strong skipping.

Numerical Results

At last, we show the benefits of the new filtering techniques and the heuristic enhancements that we just introduced in practice. For our experiments, we use the standard benchmark generator by David Pisinger (Pisinger 2005) with two slight modifications: First, we changed the code so that it can generate instances with very large profits and capacities in $[1, 10^7]$. This change was necessary as a small range of profits and weights obviously plays into the hands of our new algorithm, especially when considering problem instances with many items. Secondly, we ensure that the code actually generates a series of different instances by setting a new seed for the random number generator each time a new instance is generated (the original code produces a series of instances with the same profits and weights and only changes the relative capacity of the knapsack when the range of profits and weights stays fixed). These changes were made to ensure a fair and sound comparison. We generate random knapsack instances according to five different input distributions. In all cases, profits and weights are drawn from $[1, 10^7]$. For uncorrelated instances, profits and weights are drawn independently. Weakly correlated instances are generated by choosing profits $p_i \in [w_i - 10^6, w_i + 10^6]$. Instances are strongly correlated when setting $p_i = w_i + 10^6$. In almost strongly correlated instances, profits are uniformly drawn from $[w_i + 10^6 - 10^4, w_i + 10^6 + 10^4]$. A subset sum instance is characterized by $p_i = w_i$ for all items.

All experiments were run on a 2 GHz Intel Pentium M Processor with 1 GByte main memory. In Table 1, we compare our pure algorithm (SL) with that of (Fahle & Sellmann 2002). On instances where the longest increasing weight sequences are in $o(n)$ when items are ordered according to increasing or decreasing efficiency, we see clearly the benefits of the new propagator: On subset sum instances, the effort of our algorithm is practically constant and, as the input size grows, our algorithm is easily several orders of magnitude faster. Moreover, on uncorrelated and weakly correlated instances, we achieve some decent speedups on larger instances. At the same time, on strongly and almost strongly correlated instances, that are less favorable for our filtering algorithm, we still see some slight performance gain.

In the same table, we also show the performance gains by using the early termination heuristic (SL-ET) which gives us roughly another factor of two pretty consistently. When using strong skipping on top of early termination (not shown in the table), we do not achieve significant further savings. This is easily explained by the fact that $\tilde{w}_i^{(l)}$ will be almost equal to w_i when using the early termination heuristic. The situation changes, however, when using strong skipping *without* early termination (SL-SS): We see another dramatic performance gain that leads to an almost equally great performance over all input distributions (except for the degenerated case of subset sum instances). Eventually, SL-SS beats the algorithm from (Fahle & Sellmann 2002) by up to two orders of magnitude!

In Table 2, we give more insight into the inner workings of the contesting algorithms. The original linear-time algorithm often considers significantly more items for filtering than our propagation algorithms, which is what we predicted and which is, of course, the reason why our algorithms run faster. What is remarkable is to which extent the strong skipping heuristic is able to reduce the workload further, especially on strongly and almost strongly correlated instances: For large input sizes, less than 1% of the items need to be considered, and this reduction in workload can almost entirely be translated into a reduction of CPU-time. It is subject to further analysis to explain this effect theoretically.

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.

Katriel, I., and Van Hentenryck, P. 2006. Randomized filtering algorithms. Technical Report CS-06-09, Brown U.

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

Sellmann, M. 2003. Approximated consistency for knapsack constraints. In *CP'03*.

# Items		1,000				10,000				100,000			
Algorithm		Lin	SL	SL-ET	SL-SS	Lin	SL	SL-ET	SL-SS	Lin	SL	SL-ET	SL-SS
Uncorrelated	1%	73	31	21	9.4	722	143	90	14	7.3K	1.1K	515	104
	2%	74	33	20	6.0	729	156	89	14	7.3K	1.2K	513	103
	5%	75	36	19	4.5	736	182	93	13	7.4K	1.6K	531	99
	10%	74	40	19	3.8	732	206	94	12	7.4K	2.0K	567	92
Weakly Correlated	1%	81	30	17	5.3	792	144	74	15	8.0K	1.0K	381	105
	2%	85	33	15	5.2	827	164	73	13	8.3K	1.3K	379	103
	5%	89	38	15	3.3	872	192	75	14	8.8K	1.7K	381	98
	10%	92	43	15	3.5	899	209	74	12	9.1K	2.0K	389	90
Srongly Correlated	1%	81	68	31	5.2	791	654	260	14	8.1K	7.0K	2.6K	103
	2%	84	67	29	3.7	820	653	262	14	8.2K	7.0K	2.7K	101
	5%	88	66	29	5.0	867	643	257	12	8.7K	6.9K	2.7K	96
	10%	91	64	27	4.0	897	626	246	12	9.1K	6.6K	2.6K	86
Alm.Strongly Correlated	1%	80	52	29	5.6	792	296	165	14	8.0K	2.8K	1.4K	104
	2%	84	52	28	4.3	822	308	167	13	8.3K	3.1K	1.4K	101
	5%	88	53	27	3.7	867	324	167	12	8.8K	3.3K	1.4K	97
	10%	91	53	27	4.1	897	331	166	12	9.1K	3.4K	1.4K	87
Subset Sum	1%	59	2.2	1.4	2.1	590	2.1	1.0	3.0	6.1K	3.8	1.3	2.8
	2%	59	1.9	0.8	2.1	591	2.5	1.7	2.0	6.0K	3.7	1.3	3.2
	5%	59	2.3	0.8	2.0	592	2.9	1.1	3.4	6.0K	2.9	2.0	3.5
	10%	60	2.1	1.0	2.1	979	2.6	1.5	1.8	6.1K	3.0	1.6	2.7

Table 1: Knapsack filtering in the presence of a lower bound with 1%, 2%, 5%, or 10% gap to the linear relaxation. We report the average CPU-time [μ-sec] of the Fahle-Sellmann algorithm (Lin), our sublinear algorithm without heuristic enhancements (SL), with early termination (SL-ET), and with strong skipping (SL-SS) on benchmark sets with 100 random instances each that were generated with respect to one of five different input distributions: uncorrelated, weakly correlated, strongly correlated, almost strongly correlated, or subset sum.

# Items		1,000				10,000				100,000			
Eff. | Algo.		*Filt.*	Lin	SL/SL-ET	SL-SS	*Filt.*	Lin	SL/SL-ET	SL-SS	*Filt.*	Lin	SL/SL-ET	SL-SS
Uncorrelated	LIS			219				779				2.6k	
	1%	*40*	986	161	52	*97*	9.8K	512	104	*977*	99K	2.3K	982
	2%	*24*	975	147	33	*94*	9.7K	514	100	*957*	97K	2.3K	963
	5%	*13*	946	139	19	*88*	9.4K	517	93	*901*	94K	2.3K	907
	10%	*8*	906	138	13	*78*	9.0K	517	83	*811*	90K	2.2K	817
Weakly Correlated	LIS			171				612				3.5k	
	1%	*24*	996	127	31	*97*	9.9K	441	103	*977*	99K	2.1K	982
	2%	*16*	991	120	22	*95*	9.9K	440	100	*957*	99K	2.1K	962
	5%	*9*	977	114	14	*89*	9.8K	437	94	*897*	98K	2.0K	902
	10%	*7*	956	114	12	*78*	9.5K	430	83	*801*	95K	2.0K	806
Strongly Correlated	LIS			1k				10k				100k	
	1%	*25*	997	684	28	*97*	10.0K	6.7K	100	*975*	100K	66K	978
	2%	*17*	994	673	20	*94*	9.9K	6.6K	97	*954*	99K	66K	957
	5%	*11*	982	656	11	*88*	9.8K	6.5K	91	*895*	98K	65K	898
	10%	*7*	963	633	10	*78*	9.6K	6.3K	81	*799*	96K	63K	802
Alm.Strongly Correlated	LIS			545				2.3k				8.3k	
	1%	*25*	997	375	29	*96*	10.0K	1.4K	100	*976*	100K	5.4K	979
	2%	*17*	994	367	20	*94*	9.9K	1.4K	97	*956*	99K	5.4K	959
	5%	*10*	982	358	13	*88*	9.8K	1.4K	91	*900*	98K	5.3K	903
	10%	*7*	963	353	10	*78*	9.6K	1.4K	81	*808*	96K	5.2K	811
Subset Sum	LIS			2				2				2	
	1%	*0*	279	2.0	2.0	*0*	2.8K	2.0	2.0	*0*	28K	2.0	2.0
	2%	*0*	283	2.0	2.0	*0*	2.8K	2.0	2.0	*0*	28K	2.0	2.0
	5%	*0*	296	2.0	2.0	*0*	3.0K	2.0	2.0	*0*	30K	2.0	2.0
	10%	*0*	317	2.0	2.0	*0*	3.2K	2.0	2.0	*0*	32K	2.0	2.0

Table 2: Average number of items considered by each algorithm for the same benchmark problems as in Table 1. We also report the number of items that were filtered (Filt.), and the average sum of longest increasing subsequences when considering items in order of increasing and decreasing efficiency (LIS).