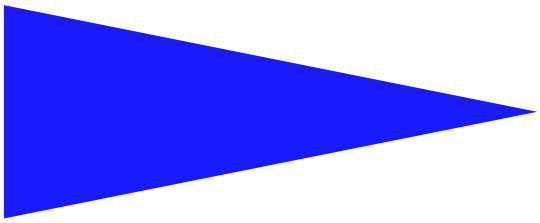
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° PI-1152



UNBOUNDED KNAPSACK PROBLEM: DYNAMIC PROGRAMMING REVISITED

RUMEN ANDONOV AND VINCENT POIRRIEZ AND SANJAY RAJOPADHYE

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Unbounded Knapsack Problem: Dynamic Programming Revisited

Rumen Andonov* and Vincent Poirriez** and Sanjay Rajopadhye***

Thème 1 — Réseaux et systèmes
Projet API

Publication interne n° PI-1152 — Octobre 1997 — 24 pages

Abstract: We present EDUK, an efficient dynamic programming algorithm for the unbounded knapsack problem (UKP), a classic NP-hard combinatorial optimization problem. It is based primarilty on a new and useful *dominance* relations between object types called *threshold dominance*. We demonstrate that threshold dominance is a strict generalization of all the previously known dominance relations. We then show that combining it with a sparse representation of the iteration domain and the periodicity property leads to a drastic reduction of the solution space. Numerous computational experiments with various data instances are presented to validate our ideas and demonstrate their efficiency. We also compare EDUK with the available and widely used exact algorithm MTU2.

Key-words: Integer Programming, Dominances, Dynamic Programming, periodiciy, cominatorial optimization

(Résumé : tsvp)

* Rumen.Andonov@univ-valenciennes.fr
** Vincent.Poirriez@univ-valenciennes.fr
*** Sanjay.Rajopadhye@irisa.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Le problème de sac à dos non borné : la programmation dynamique retour à la programmation dynamique

Résumé :

On présente EDUK, un algorithme efficace de programmation dynamique pour le problème du sac à dos non borné. Il repose sur une nouvelle relation de dominances entre les types d'objets, appelée la *dominance de seuil*. On montre que cette dominance est une généralisation des différentes dominance connus dans la littérature. On montre ensuite que sa combinaison avec une représentation creuse de l'espace d'itération et la propriété de périodicité donne une réduction considérable de l'espace de recherche de solution. De nombreuses expériences avec divers jeux de données valident nos idées et montrent l'efficacité d'EDUK. On compare aussi EDUK et MTU2 un algorithme connu dans la littérature.

Mots clés : programmation en nombre entiers, dominance, programmation dynamique, périodicité, optimisation combinatoire

1 Introduction

The unbounded knapsack problem (UKP) is a classic NP-*hard* combinatorial optimization problem with a wide range of applications [6, 10, 12]. It may be formulated as follows: we are given a knapsack of capacity c, into which we may put m types of objects. Each object of type i has a *profit*, p_i, and a *weight*, w_i, (w_i, p_i, m and c are all positive integers, and we have an *unbounded* number of copies of each object type). Determine, for $i = 1 \ldots m$, the number x_i, of i-th type objects to be chosen so as to maximize the total profit without exceeding the capacity, i.e.,

$$\max \left\{ \sum_{i=1}^{m} p_i x_i \ : \ \sum_{i=1}^{m} w_i x_i \leq c, \ x_i \geq 0 \text{ integer}, \ i = 1, 2, \ldots, m \right\} \tag{1}$$

The two classic approaches for solving this problem exactly are branch and bound (B&B) [12] and dynamic programming (DP) [3, 6, 10], the subject of this paper. It takes $O(mc)$ time, and works in two phases: in the *forward* phase, we calculate the *optimal value* of the profit function (by filling up an $m \times c$ table), and then we use this in the *backtracking* phase to determine an actual solution which has this optimal profit.

In 1963, Gilmore and Gomory proposed the notion of *dominance* [7]. Intuitively, if an object type has a larger (or no smaller) weight, and smaller (or no larger) profit than another, the former will never occur in an optimal solution (recall that we have an unbounded number of copies). Martello and Toth introduce a more powerful dominance [12], based on the fact that an object type can be dominated by *multiple copies* of another. In both cases, a dominated object type can be discarded from consideration. Often, this is detected by preprocessing the inputs. Dudzinski observes that dominance is a partial order [5] and proposes efficient preprocessing techniques. All previous work on dominance (well surveyed by Pisinger [13]) is in the context of branch and bound algorithms (or at least as a preprocessing step for other algorithms). Exploiting dominance (also called coefficient reduction) is very important. Empirical evidence shows that the number of non dominated items is usually extremely small. For uniform distribution of the weights and profits, this has been analytically confirmed [11], which actually raises questions about the suitability of the "uncorrelated" data sets [12].

Another well known approach to reduce the search space is available for dynamic programming based algorithms. In 1966 Gilmore and Gomory gave a *periodicity* property [8] which allows significant reduction of the *capacity* dimension of the search space.

A third technique, again applicable for dynamic programming was recently presented by Andonov and Rajopadhye [2]. It is based on the fact that the standard recurrence is monotonic. This allows a "sparse" representation of the computation. A similar property also holds for the 0/1 knapsack problem, and an algorithm using a sparse representation (LIFO lists) was proposed at least twenty years ago [9], but this had not previously been used for the unbounded knapsack problem.

In this paper, we use the notion of ***collective dominance***, where an object type is dominated by (a subset of) the other object types, i.e., there is a linear combination of these object types which *collectively* has a larger profit and lower weight. Then too, the former can be discarded. This observation was also made independently by Pisinger [13]. However, detecting such a dominance necessitates the solution of a knapsack (sub) problem, and it is unacceptable as a preprocessing step.

Our first key idea is that collective dominance could be naturally exploited in a dynamic programming algorithm because the solutions of all knapsack sub problems of smaller capacity (and the same set of object types) are available "for free".

Our second key idea is based on our empirical observation that even among object types not dominated collectively by the others, many contribute *very few times* in the optimal solution. Note that every such object type contributes at least once, namely for the (sub) problem with capacity equal to its weight. This leads to the notion of ***threshold dominance***, which occurs when a linear combination of the other object types collectively has a larger profit and lower weight than a *finite number*, say α, instances of the object type. Hence no optimal solution will have more than α copies of this object type. As a result, we can discard this object type from consideration beyond a *threshold* capacity, namely α times its weight.

Our goal is to exploit *all* these properties (collective and threshold dominance, sparse representation, periodicity, and efficient backtracking). Our main contributions are as follows.

- We introduce and formalize the notions of ***collective dominance*** and ***threshold dominance***. We present a complete classification and show that threshold dominance strictly includes collective dominance, which is a strict generalization of the previously studied versions of dominance. We show that collective dominance is *maximal*, in terms of coefficient reduction.

- We also show that threshold dominance enables easy testing for periodicity, in the sense that periodicity is attained if and only if the cardinality of the set of object types not dominated beyond a given threshold is 1.

- We design a dynamic programming algorithm which incorporates these properties into a *slice wise* evaluation of a well known (but not commonly used) recurrence for the forward phase. This eliminates preprocessing, which is incorporated into the main algorithm itself. Furthermore, we adapt the sparse representation and algorithm [2] to this recurrence.

- We present a backtracking algorithm with the same performance Hu's well know idea of computing an auxiliary recurrence [10], but *without any* auxiliary information. In addition, our technique enables us to determine *all* solutions.

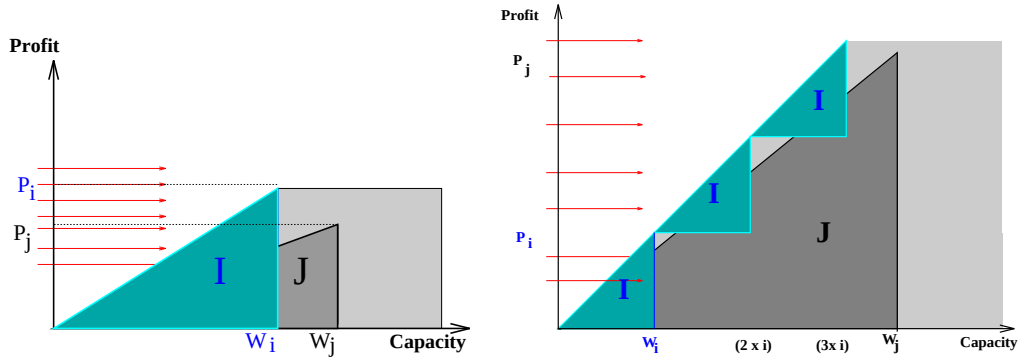
- We validate our algorithm by numerous computational experiments with a number of data sets:
  - the standard random (uncorrelated, and weakly and strongly correlated) data sets used in the literature [12, 5];
  - similar sets but with significantly larger weight interval (in order to preclude the "false" advantages due to simple dominance);
  - ***realistically random*** data sets, in which no object type is simply dominated by (i.e., has higher weight and smaller profit than) another, since such a situation is almost inconceivable in real life;
  - "hard instances" generated either by analytical formulæ or randomly by filtering the runs of hundreds of examples;
  - data drawn from real life cutting stock problems.

  We had excellent performance (even for very large instances of the UKP). We are still searching for problems that are difficult for our algorithm, whether artificially generated (some of which are illustrated here) or from real data.

Our implementation, first described in [1], preserves two useful properties of dynamic programming: (i) knowledge of the solution for *any* capacity lower than the given capacity; and (ii) ability of *reuse* the known solutions for solving *larger* capacity problems. The remainder of this paper is organized as follows. In Section 2 we develop collective and threshold dominance, give their properties and their relation with periodicity. Section 3 describes how our algorithm by slices incorporates all the above features and also presents the backtracking algorithm. Section 4 presents our experimental results, and we conclude in Section 5.

Notation: $\mathcal{Z}_+$ denotes the set of non negative integers. We use the m-vectors, $\vec{w}$ and $\vec{p}$, respectively, to denote the weights and profits for a given problem instance. An object type can be identified either by its index, i, or by the ordered pair (w_i, p_i). $M = \{1, \ldots, m\}$ denotes the set of all object types. Any m-vector $\vec{x}$, in $\mathcal{Z}_+^m$, may be viewed as a candidate solution, whose i-th element, x_i specifies the number of chosen instances of the i-th object type. For any candidate solution $\vec{x}$, $W(\vec{x}) = \sum_{i \in M} x_i w_i$ and $P(\vec{x}) = \sum_{i \in M} x_i p_i$ are called *its* **weight** and **profit**. We denote by UKP_j^S a knapsack (sub)-problem with capacity j and whose object types are restricted to $S \subseteq M$; the original problem (1) is thus UKP_c^M. $F(j, S)$ is the optimal profit that can be achieved for knapsack UKP_j^S.

Definition 1 We associate to each candidate solution, $\vec{x}$, a **solution point**, namely the pair, $(s, f) = (W(\vec{x}), P(\vec{x}))$. We say that $\vec{x}$ is a **feasible solution** for UKP_c^S if $\forall i \in M \backslash S$, $x_i = 0$ (the operator $\backslash$ represents set difference), and if its weight, $W(\vec{x})$ is no greater than the knapsack capacity, c.



Simple dominance $j \ll_s i$ multiple dominance $j \ll_m i$

Figure 1: Illustration of the two pairwise dominance relations: simple and multiple. An object type is shown as a triangle of width w and height p.

Definition 2 Any feasible solution $\vec{x}$ for UKP_c^S such that $P(\vec{x}) = F(c, S)$ is said to be **an optimal solution** for UKP_c^S. $\mathrm{Opt}(\mathrm{UKP}_c^S)$ denotes the set of the optimal solutions of UKP_c^S.

Definition 3 The ***profitability*** of an object type is the ratio of its profit to weight, $\frac{p_i}{w_i}$. Two object types with equal profitability are said to be ***equiprofitable***, $i \equiv j$.

In certain algorithms, we may sort the object types by order of profitability, i.e., the total order relation in M^2 defined as $j \sqsubseteq k$ iff either $\frac{p_j}{w_j} < \frac{p_k}{w_k}$ or $(\frac{p_j}{w_j} = \frac{p_k}{w_k}$ and $w_j \geq w_k)$. This order is often used in the literature, typically in a preprocessing phase either on all the object types [6, 12] or on pre-selected core problems [5, 12].

Let b be the index of the ***best object type*** (the one with greatest profitability).

2 Dominance Relations

We now define the different dominance relations and describe their properties and interactions. We first introduce dominance between solution points (actually, any two pairs of integers), and then describe dominances between object types. Next, we give the strict inclusion relations between them, and finally describe how periodicity can be detected using threshold dominance.

Definition 4 A pair, (s', f') **dominates** another, (s, f), denoted as $(s, f) \lhd (s', f')$, iff $s \geq s'$ and $f \leq f'$.

i is **simply dominated** by j, written as $i \ll_s j$, iff $(w_i, p_i) \lhd (w_j, p_j)$.

i is **multiply (simply) dominated** by j, written as $i \ll_m j$, iff $\left\lfloor \frac{w_i}{w_j} \right\rfloor \geq \frac{p_i}{p_j}$

These two relations (called *pairwise* dominances) are illustrated in Fig 1. It is easy to verify that $\lhd$ is a partial order (as noted by Dudzinski for the $\ll_m$ relation [5]). Furthermore,

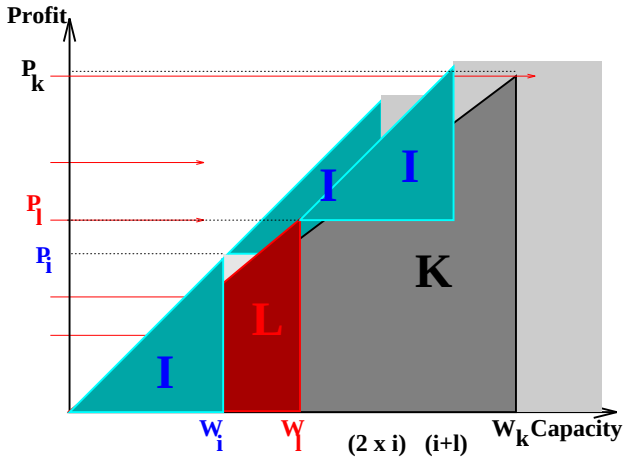


Figure 2: Collective (maximal) dominance: $k \ll \{i, l\}$

if, for some $\vec{x}$ and $\vec{x'}$, feasible solutions of respectively UKP_c^T and $\mathrm{UKP}_{c'}^{T'}$, their respective *solution points* satisfy $(s, f) \lhd (s', f')$, then either $\vec{x}$ cannot be an optimal solution, for *any* capacity, and for any subproblem which includes at least $T \cup T'$, or $f = f'$ and so $\vec{x'}$ is also optimal.

Definition 5 Let J be a set of object types (i.e., $J \subset M$) and $i \notin J$. The i-th object type is **collectively dominated** by J, written as $i \ll J$ iff $\exists \vec{x} \in \mathcal{Z}_+^m \mid (\sum_{j \in J} x_j w_j \leq w_i$, and $\sum_{j \in J} x_j p_j \geq p_i)$. In other words, a positive linear combination of the objects in J yields a "better" (or at least, no worse) solution than i.

$\Omega = \{i \in M \mid i \not\ll M \backslash \{i\}\}$ is the set of object types that are not collectively dominated. $w_{\max}^{\Omega}$ is the weight of the heaviest object type in Ω.

Collective dominance (illustrated in Fig. 2) is maximal in the following sense: if $i \ll J$, the i-th object type can be discarded from consideration without changing the optimal value for *any* capacity, c. Conversely, whenever an object type i is in Ω, it contributes to the optimal solution for some capacity (in particular, for $c = w_i$).

Finally, we observed empirically that, many object types contribute relatively few times to the optimal solution (for any capacity). The explanation lies in the fact that for each of these object types, there is a threshold, beyond which it can be substituted by a linear combination of other objects. This led us to the following definition.

Definition 6 Let $J \subseteq M$, $i \notin J$, $t > 0$ and consider $t' = \alpha w_i = \left(\lfloor \frac{t}{w_i} \rfloor + 1 \right) w_i$, the smallest multiple of w_i strictly greater than t. We say that the i-th object type is **dominated above the threshold** t by J, written as $i \ll^t J$ iff $\exists \vec{x} \in \mathcal{Z}_+^m \mid \sum_{j \in J} x_j w_j \leq t'$, and $\sum_{j \in J} x_j p_j \geq \alpha p_i$.

Note that if $j \equiv i$ there exists a t (namely $l - 1$ where l is the lcm of w_i and w_j), such that $i \ll^t \{j\}$ and $j \ll^t \{i\}$. To avoid ambiguity, we consider that only the object with the larger weight is dominated above t by the other. Threshold dominance is illustrated in Fig. 3.

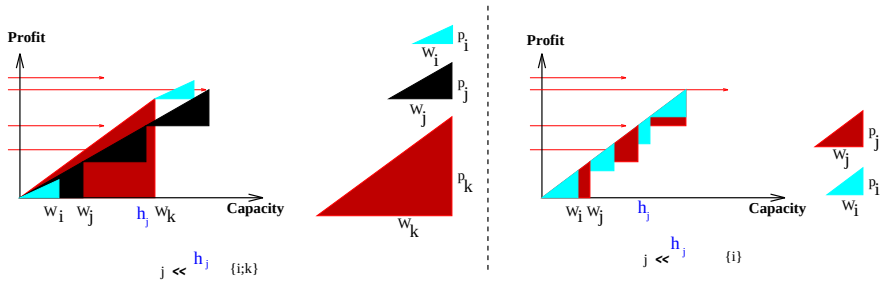


Figure 3: Illustration of threshold dominance. (on the left:) object type j is dominated by $\{i,k\}$ above $2w_j$, and (on the right:) a particular case, when two object types have the same interest (right); here, $j \ll^{3w_j} \{i\}$.

Observe that if $i \ll^t J$, then for any positive integer, k, $i \ll^{t+k} J$ and for $t' = \lfloor \frac{t}{w_i} \rfloor w_i$, $i \ll^{t'} J$. Hence, we may define **the threshold** h_i of the i-th object type as the smallest multiple of its weight such that $i \ll^{h_i} M\backslash\{i\}$ if such an h_i exists. Formally, let $H_i = \{t \mid i \ll^t M\backslash\{i\}\}$. Then, h_i is the minimum of H_i (if H_i is empty, $h_i = \infty$).

Proposition 1 If $i \ll^t J$, for any $c > t$, there exists $\vec{x'} \in \mathrm{Opt}(\mathrm{UKP}_c^M)$ such that $x'_i w_i < t$.

Proof: Recall that $t' = \alpha w_i = \left(\lfloor \frac{t}{w_i} \rfloor + 1\right) w_i$ is the smallest multiple of w_i strictly greater than t. By definition of threshold dominance, $\exists \vec{z} \ : \sum_{j\in J} z_j w_j \leq \alpha w_i$ and $\sum_{j\in J} z_j p_j \geq \alpha p_i$. Let $\vec{x}$ be a solution vector in $\mathrm{Opt}(\mathrm{UKP}_c^M)$. If $x_i < \alpha$ then $\vec{x}$ is the desired solution. Otherwise let $x_i = k\alpha + r$ for positive integers k, r such that $0 \leq r < \alpha$. We construct the desired optimal solution by "replacing" the $k\alpha$ copies of the object type, i by k copies (of the part corresponding to the object types in J) of $\vec{z}$. Thus the vector $\vec{x'}$ given by

$$x'_j = \begin{cases} x'_j = x_j & \text{if } j \in M\backslash\{i\}\backslash J \\ x'_j = x_j + kz_i & \text{if } j \in J \\ x'_j = r & \text{if } j = i \end{cases}$$

is the desired optimal solution. ∎

Hence if $i \ll^t J$ the i-th object may be discarded beyond a capacity of t. The threshold h_i of an object type is thus the largest multiple, α, of its weight, for which the optimal solution $\vec{x}$ of $\mathrm{UKP}_{h_i}^M$) is unique and given by $x_k = 0, k \neq i$ and $x_i = \alpha$. This gives a sufficient condition to detect the threshold.

Proposition 2 If $y \bmod w_i = 0$ and $\exists \vec{x} \in \mathrm{Opt}(\mathrm{UKP}_y^M)$ such that $x_i < y/w_i$ then $h_i < y$.

Proof: A feasible solution for UKP_y^M consists of just y/w_i copies of i, and has the profit yp_i/w_i. Since $\vec{x} \in \mathrm{Opt}(\mathrm{UKP}_y^M)$, we have $x_i p_i + \sum_{j\in M\backslash\{i\}} x_j p_j \geq yp_i/w_i$ and

$\sum_{j \in M \setminus \{i\}} x_j w_j \leq y - x_i w_i \leq y$. Therefore the object type i is dominated above $t = y - x_i w_i$ and hence $h_i < y$. ∎

We now present some useful properties the dominances and as well as some relationships between them.

1. i is collectively dominated by $\{j\}$ iff $i \neq j$ and i is multiply dominated by j.
2. i is collectively dominated by the set J iff i is threshold dominated above 0 by J.
3. The object type with the smallest weight is never collectively dominated.
4. There is no threshold beyond which the best object type is threshold dominated.
5. If i is threshold dominated above some t by a set J then there exists at least one object type $j \in J$ such that $i \sqsubseteq j$.

The above properties lead to the following result.

Theorem 1 The four dominance relations are in strictly increasing order of generality.

Note that the cost required to check for a dominance relation is increasing from $\ll_s$ to $\ll^t$. In fact, $\ll$ and $\ll^t$ require the solution of knapsack sub-problems.

2.1 Periodicity

We now describe the relation between threshold dominance and periodicity, a well known property of the UKP [6, 8]. Periodicity states that beyond a capacity, y, *only the best object type contributes again* to the solution. Hence, knowing the solutions for each capacity below y is sufficient to solve the problem for any capacity. Formally it can be formulated as follows:

$$\exists y \text{ such that } \forall c \geq y, F(c, M) = \lceil (c-y)/w_b \rceil p_b + F(c - \lceil (c-y)/w_b \rceil w_b, M) \qquad (2)$$

We define the *period level*, $y^\star$ as the smallest y satisfying (2). The following result relates the periodicity property with the threshold dominance and gives a simple test to check if the period level is reached. For any capacity y, we define $\mathcal{U}(y) = \{j \in M \mid j \not\ll^y M\}$ as the set of object types that are not dominated above a threshold y.

Theorem 2 The set $\mathcal{U}(y)$ is a singleton, iff the capacity y satisfies Eqn. 2.

Proof:

$\Rightarrow$ Obvious.

$\Leftarrow$ Let k be an object type different from the best object type. We will show that $k \ll^y M \setminus \{k\}$.

Let t' be the smallest multiple of w_k such that $t' > y$. According to Eqn. (2) there exists $\vec{x} \in \text{Opt}(\text{UKP}_{t'}^{M})$ such that $x_b > 0$ which implies $x_k < y/w_k$. Hence from proposition 2 we obtain $h_k < y$. Therefore only the best object type is not dominated above y. ∎

Corollary 1 $y^\star = min\{y \text{ such that } \mid \mathcal{U}(y) \mid = 1\}$

3 An Efficient Implementation

We now describe how the above properties are combined into an efficient algorithm, EDUK (**E**fficient **D**ynamic programming for the **U**nbounded **K**napsack problem). The basic recurrence for the forward phase is given below. It is a standard, though not common recurrence, and can be easily derived from the principle of optimality. $F(c, M) = g(c, m)$ where,

$$g(j,k) = \begin{cases} j < 0 \text{ and } k = m & \Rightarrow & -\infty \\ j = 0 \text{ or } k = 0 & \Rightarrow & 0 \\ j > 0 & \Rightarrow & \max(g(j, k-1), g(j - w_k, m) + p_k) \end{cases} \quad (3)$$

This is viewed as an $m \times c$ table, having m columns, one for each object type. The columns have height c and are "filled from top to bottom" (origin at the top-left). If and when we detect that an object type is (collectively) dominated by the other remaining object types, its column is deleted, thus achieving coefficient reduction. If and when we detect that the capacity has exceeded the threshold beyond which an object type does not contribute any more, the *remainder* of its column is deleted, thus achieving capacity reduction. And finally, we also exploit *sparsity*: in the (sub) columns where the recurrence is to be evaluated, we do not evaluate it *for all values of* j. Since the function $g(j, k)$ is monotonically increasing in j, we need to compute it *only when it changes value* (at the so called critical points).

Our challenges are to detect dominance (collective as well as threshold) as (i) cheaply and (ii) early as possible, and (iii) to integrate it with the sparse algorithm, i.e., the data required for this detection must be available only by inspecting the results at critical points.

The recurrence is evaluated by horizontal slices. This can be done easily if the last column of the table is retained up to the start of the slice (say capacity c_0), and the boundary conditions of Eqn. 3 are modified appropriately. The height of each slice could be constant (a parameter of the algorithm) or variable. For each slice, EDUK maintains $g_0 = F(c_0, M)$, the value of $\text{Opt}(\text{UKP}_{c_0}^{M})$ and the index of the current best object, b'. The main data structures are three lists:

- $\mathcal{R}$, a list of *residual* object types. It is initialized to M, and (at the start of any slice) contains the set of object types whose weights are larger than c_0 (hence we don't yet know whether they are collectively dominated or not), and which have not been discarded by some "easy" tests (described later).
- $\mathcal{I}$, a list of size p, a parameter, which contains the p object types *introduced* during this slice (i.e., instead of constant-height slices, we have slices of variable height, but with exactly p new object types introduced in each slice).
- $\mathcal{U}$, the list of "currently non-dominated" object types (at the start of each slice). These are object types that (i) have weights smaller than c_0, (ii) are not collectively dominated by the other object types, and whose (iii) thresholds are not less than c_0.

Evaluation of a slice: The computation performed in evaluating a slice consists of the following.

1. **Pre-processing:** Make a single pass through $\mathcal{R}$ in order to determine the p lightest object types to introduce in the slice. The list $\mathcal{I}$ is maintained sorted by increasing weight. For each element from $\mathcal{R}$, EDUK perform two simple tests: test if it is collectively dominated by g_0, and test whether it is multiply dominated by the best object type so far. It is a tradeoff whether this test, which involves a couple of arithmetic operations should be done here for *all* elements of $\mathcal{R}$, or in the post-processing step for only a few object types. If so it is immediately discarded. If it passes these tests, insert it in $\mathcal{I}$, and while doing so, check that it is not simply dominated by the object types already there. Then too it is discarded (these are the "easy tests" mentioned above).

2. **Main:** For each i in $\mathcal{I}$, resolve the successive knapsack sub-problems of capacity w_i, using only object types in $\mathcal{U}$. This is done by evaluating the recurrence $g(j,k)$, using the sparse algorithm (see Sec. 3.1). At the end of each such evaluation, check if the object type i is collectively dominated by the others. If so, it is discarded, otherwise it is appended at the end of $\mathcal{U}$.

3. **Post-processing** Update the current best object type (using those elements of $\mathcal{I}$ that were added to $\mathcal{U}$ in step 2 above. Also update the current optimal value, $(w_p, F(w_p, M))$. Finally, for all object types in $\mathcal{U}$, test to see if their threshold has passed, and if so, remove them from $\mathcal{U}$. If $\mathcal{U}$ is sorted in decreasing order of profitability, it is possible to detect threshold dominance, at the earliest possible moment, and this may lead to some savings (this implementation tradeoff is not discussed here).

Standard Phase The slice-wise evaluation is performed repeatedly until $\mathcal{R}$ becomes empty. This concludes the *reduction phase*. Assuming that the capacity c and the period level have not yet been reached (otherwise we are done), we move on to the *standard phase*. All that remains is to continue to evaluate the recurrence, and this done with *constant-height* slices (but without the preprocessing above). As before, we detect threshold dominance at the end of each slice, and thus $\mathcal{U}$ may continue to decrease. Each time this occurs we test to see if $\mathcal{U}$ is a singleton. When this happens, we know that we have attained periodicity, and the solution is computed with the closed form formula of Eqn. 2, and we are done!

3.1 A sparse representation

Yet another method of search space reduction is the so called ***sparse representation*** [2]. A complete discussion is beyond the scope of this paper, and we simply present an overview of the method here. It is based on the key observation that the function $g(j,k)$ is monotonically

increasing in j. As a result, there is no need to compute the recurrence for *all* values of j (i.e., points in a column). It suffices to calculate it at only those j indices where its value changes. If this is done, the value of $g(j,k)$ over the entire k-th column can be represented as a sequence of pairs (s,f), similar to the representation used to store and manipulate sparse matrices (hence the name). The sparse representation, which is fully exploited in our algorithm requires *lazy* data structures known in functional languages as lazy lists, (or streams). It is for this reason that we have chosen to implement our algorithm in a programming language in the functional style (although an implementation in C, Fortran or a more conventional language could arguably have been more efficient, we found that the program development time was considerably reduced).

3.2 Backtracking phase

In our entire discussion so far, we have not mentioned the *backtracking* phase of the dynamic programming algorithm, which consists of determining the actual solution, $\vec{x}$, once the recurrence $g(j,k)$ has been evaluated. A common criticism of DP is that like its worst case running time, its *space complexity* is claimed to be $\Theta(mc)$, i.e., the entire table has to be saved in order to compute the solution. Often overlooked is the fact that, for the *unbounded* knapsack problem, we can compute an auxiliary recurrence during the forward phase, and then, backtracking can be performed by traversing only the *last* column of the table. This was initially presented in Hu's text in 1969 [10]. Another algorithm was presented by Garfinkel and Nemhauser [6]. Both algorithms use an auxiliary function during the forward phase and thus two lists of size c are kept in the memory. We now present an algorithm which achieves the same space and time complexity as these, but with *no* overhead in the forward phase. In addition, our technique enables us to determine *all* solutions, unlike the others.

We start with recursions and notations similar to these of Garfinkel and Nemhauser [6], but we use them in a slightly different manner and also take into account the notion of dominance.

Proposition 3 Let $\mathcal{D}(y) = \{k : x_k > 0 \text{ in some } \vec{x} \in \mathrm{Opt}(\mathrm{UKP}_y^M)\}$. Then when $F(y,M) \neq 0$ and for some $k \in M$ such that $y - w_k \geq 0$ we have

$$F(y,M) - p_k = F(y - w_k, M) \Leftrightarrow k \in \mathcal{D}(y) \tag{4}$$

Recall that $\Omega = \{j \in M \mid j \not\ll M\backslash\{j\}\}$ and let us initialize $S := \Omega$. We then have the following algorithm where the vector $\vec{x}$ is initialized to $\vec{0}$.

```
while S <> empty and y > 0 do
  let k in S
  if y-w_k < 0 then  S:= S-{k}
  else if F(y-w_k,M) = F(y,M)-p_k then (x_k := x_k + 1; y := y - w_k)
       else  S:= S-{k}
```

Clearly, the algorithm is correct if Ω were replaced by M, since it follows directly from Eqn. (4). Because collective dominance is maximal, no solution is lost when using Ω. Moreover Ω is known at the end of the forward phase and is often very small. Furthermore, the choice of k in the second line above allows us to systematically determine *all* solutions. A useful heuristic for rapidly obtaining one solution is to follow the order of the decreasing profitability of object types (i.e., the best object type to be considered first). Finally, the algorithm above can be directly adapted to exploit sparsity, and this is implemented in the working version of our algorithm.

4 Experimental Results

We now evaluate the performance of the EDUK algorithm. All experiments were run on a 64 bit DEC/Alpha 2100 5/250 machine. Our code was written in `objective caml`[1]. For comparison purposes, we also give the results of the running time (on the same machine) with the standard branch and bound algorithm, MTU2 due to Martello and Toth [12], written in `fortran 77`, available in the public domain. It is recognized as the *de facto* standard program for the UKP. Although the two are implemented differently, our results nevertheless serve to indicate the trends. Moreover, the EDUK was developed using a rigorous program development approach, where the transformations on the program were based on formal properties of the recurrences involved, and as a result the final `ocaml` program was very close to the recurrences. We expect that if the algorithm was recoded in `fortran 77`, the performance would improve by a small constant factor, since we expect that the implementation of arrays is better optimized in `fortran 77`.

Typically, three kinds of data sets have been used in the literature for comparing the performance of knapsack algorithms [12]:

- *Random* data sets, where weights and profits belong to a *uniform random distribution* (URD) within a specified range. The bulk of our results are based on such data sets. Our goal is to separately highlight the performance gains due to *each* of the properties we have presented. Our experiments highlight some limitations in current methods of generating random data sets for the UKP, and we also present results with what we call "realistic random" data sets.

- *Pathological* examples which exhibit worst case behavior. Since the knapsack problem is NP-complete, it is fairly easy to contrive such problem instances, for almost any given algorithm, strategy or heuristic (for example, Chung et al. [4] give hard instances for the branch and bound algorithm). We give formulæ to generate a family of problems

[1] `ocaml` is a language in the ML family. It was developed at INRIA, Rocquencourt, France. Documentation may be found at: `http://pauillac.inria.fr/caml`

for which the EDUK algorithm has high running times, and try to identify which of the properties contribute to an improvement.

- *Real life* examples drawn from actual problem instances. We have posted a request for data sets on the `sci.op-research` newsgroup, and have also contacted researchers in the domain. So far we have one class of such data, which we present here. The code will be shortly made publicly available at our ftp site, and we welcome additional feedback.

4.1 Random Examples

As described by Martello and Toth [12], one can generate three kinds of uniformly random data sets. In the *uncorrelated* case, the weights and profits are both chosen randomly, independent of each other. In the *strongly correlated* case, the weights are chosen randomly, but the profit is a linear (actually affine) function of the weight (i.e., $p_i = aw_i + b$ for some predetermined constants, a and b). In *weakly correlated* data sets the w_i's are random, and p_i is chosen randomly in the interval $aw_i \pm b$. The interval for weights used by Martello and Toth is $[10, 10^3]$ and for weakly and strongly correlated data sets, the constants are defined as $a = 1$ and $b = 100$.

Johnson and Khan [11] proved that for uncorrelated data sets, there are very few object types that are not multiply dominated, and that the expected value of the number of (multiply) non-dominated object types is as low as 1.6! An intuitive explanation is given in Fig 4. As a result, algorithms that exploit (at least) multiple dominance such as MTU2 and EDUK are bound to have excellent performance on such data sets, which are not therefore truly representative.

Another interesting point concerns the strongly correlated data sets. Since the profit of an object type is a function of its weight, the number of *distinct* object types is simply the range of possible weights. For the interval $[10, 10^3]$, none of the corresponding 990 object types is (simply) dominated by any other, whereas the 10 smallest object types are sufficient to (multiply) dominate all the other object types (it is easy to check that the object type of weight 10 (multiply) dominates *any* object type with weight greater or equal than 20). Now, if $m \gg 1000$, all distinct object types will be chosen with high probability, and hence, the number of non-dominated object types (for both the $\ll$ and $\ll_s$ relations) will be constant (resp. 10 and 990) independent of m. To avoid this effect, the range of possible weights should be reasonably large compared to m.

In spite of the above limitations, such data sets are widely used in the literature, and are almost *de rigeur* in comparing various algorithms for the knapsack and related problem(s). For this reason, we first present a series of experiments using these data sets, but with an effort to reduce the effects of duplicates. We then present experiments illustrating the beha-

A randomly selected object type (weight-profit pair) may be viewed as a point in the rectangle defined by $[w_{\min}, p_{\min}]$, $[w_{\min}, p_{\max}]$, $[w_{\max}, p_{\min}]$ and $[w_{\max}, p_{\max}]$ as shown. Now, if any other object type is chosen randomly, and happens to lie in the shaded region, it either dominates or is dominated by the first point.
Thus, there is high probability that uncorrelated data sets contain many object types that are (simply) dominated by some other, especially if the number of object types is reasonably large.

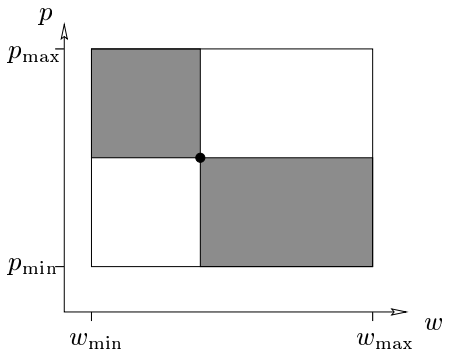


Figure 4: Why completely uncorrelated data sets are unrealistic for the UKP

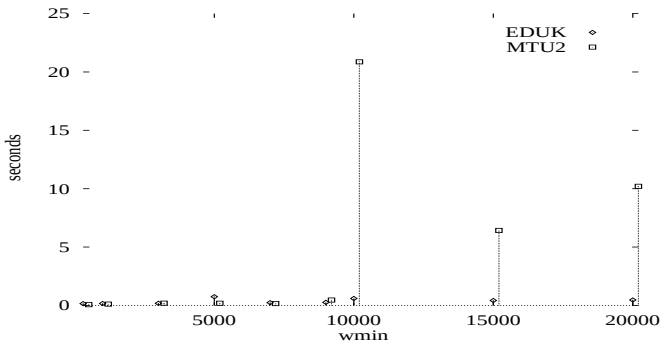
vior of both algorithms as the capacity is increased, and finally we define our ***realistically random*** data sets, and describe the performance on such examples.

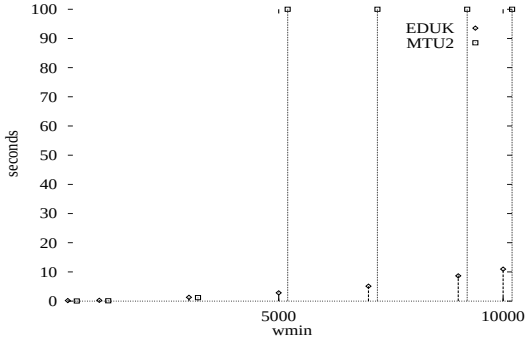
In designing our experiments, our goal was to show a progression of increasing running times and compare the performance of the two algorithms, and also to reduce the impact of duplicates. Now, the running time is very sensitive to the *number* of (multiply) non dominated object types rather than m (the explanation is the one described above: a large number of object types are dominated in uncorrelated data sets, and thus tend to be pruned out early by any reasonably smart algorithm). Pisinger recently observed [13] that the number of non dominated object types is very sensitive to $w_{\min}$: the larger the value of $w_{\min}$, the greater the number of non-dominated object types.

Based on these observations, we chose to fix m at a reasonably large value and allowed $w_{\min}$ to vary. In particular, we used the following parameters: $m = 10^5$, $w_{\max} = 10^5$, and $w_i \in [w_{\min}, w_{\max}]$, with $w_{\min}$ varying between 1 and 10^4. The capacity was fixed to a large value greater than 2×10^8. We generated the three standard data sets (uncorrelated, weakly and strongly correlated). Fig. 5 summarizes our results, showing the running times of EDUK and MTU2. The column UD gives the number of (collectively) non-dominated object types. Note that the number of multiply non-dominated object types determined by MTU2 is not available, but is no larger than this value. For the uncorrelated data sets, p_i was chosen randomly in the interval $[p_{\min}, p_{\max}]$, for the weakly correlated data sets, it was chosen randomly in the interval $[w_i - 100, w_i + 100]$, and for the strongly correlated case, it was set to $w_i + 100$. The random number generator we used was the ocaml module `random`. As noted above, EDUK is implemented in `ocaml` and MTU2 in `fortran 77`.

As can be seen, EDUK consistently outperforms MTU2, except possibly for the early part of the uncorrelated data sets. However, in this case (i) both algorithms are fast, (ii) EDUK starts having better performance as the number of non dominated object types starts to increase, and (iii) we have already argued that such data sets are not realistic. Before presenting our results with more realistic data sets, we first discuss how the running time depends on the capacity.

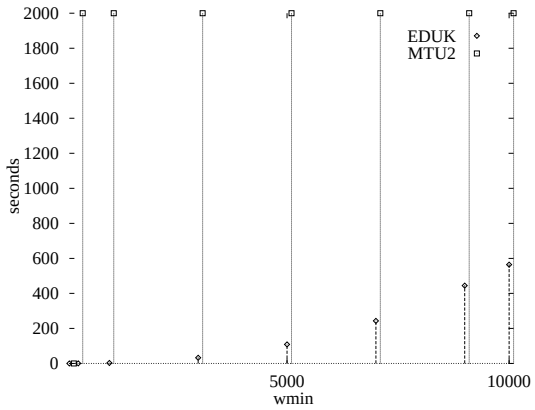
$w_{\min}$	UD	eduk	mtu2
1	1	0.150	0.062
3	1	0.151	0.062
10	2	0.153	0.064
30	3	0.150	0.061
100	4	0.156	0.060
300	4	0.155	0.061
1000	6	0.165	0.116
3000	8	0.180	0.193
10000	9	5.638	21.030
15000	7	0.425	6.426
20000	17	0.466	10.206





$w_{\min}$	UD	eduk	mtu2
1	1	0.161	0.065
3	1	0.171	0.135
10	3	0.165	0.074
30	5	0.161	0.056
100	10	0.151	0.059
300	30	0.183	0.056
1000	87	0.281	0.118
3000	260	1.293	1.191
5000	471	2.834	$1 > 2000$
7000	1084	5.123	$2 > 2000$
9000	1890	8.668	$4 > 2000$
10000	2776	10.960	$5 > 2000$

$w_{\min}$	UD	eduk	mtu2
1	1	0.150	0.053
3	2	0.158	0.053
10	6	0.158	0.059
30	18	0.161	0.058
100	64	0.168	0.062
300	187	0.313	$10 > 2000$
1000	645	2.638	$7 > 2000$
3000	1932	32.315	$8 > 2000$
5000	6026	108.516	$10 > 2000$
7000	10642	242.971	$10 > 2000$
9000	16639	444.568	$10 > 2000$
10000	23355	564.393	$7 > 2000$



Running time and the number of (collectively) non dominated object types vs $w_{\min}$, for $m = 10^5$ for uncorrelated (top), weakly correlated (middle) and strongly correlated (bottom) data sets. Each table entry (and point on the graphs) is the average of 10 runs (truncated for the *UD* column). The entry $n > 2000$ indicates that n out of the 10 problem instances took more than 2000 seconds (timed out). This happens only for MTU2, and we generously assume that the running time is 2000 seconds in this case.

Figure 5: Comparison of EDUK and MTU2 for the standard random experiments.

4.2 Sensitivity of Running time to the Capacity

Dynamic programming is often rejected out of hand for very large capacities. Hence it is important to study how the capacity affects the running time. We conducted the following

av_eduk	*min_eduk*	*max_eduk*	*av_mtu2*	*min_mtu2*	*max_mtu2*
0.088	0.083	0.100	0.074	0.039	0.122
0.083	0.066	0.116	0.049	0.038	0.133
0.086	0.066	0.100	0.047	0.037	0.056
0.091	0.083	0.100	0.051	0.040	0.122
0.105	0.083	0.116	0.054	0.040	0.140
0.094	0.066	0.116	0.048	0.039	0.086
0.094	0.083	0.116	0.082	0.040	0.178
0.082	0.066	0.100	0.068	0.040	0.126
0.094	0.083	0.133	0.093	0.040	0.226
0.095	0.083	0.116	0.066	0.040	0.102
0.091	0.066	0.116	0.063	0.037	0.226

av_eduk	*min_eduk*	*max_eduk*	*av_mtu2*	*min_mtu2*	*max_mtu2*
1.203	1.133	1.283	0.211	0.048	10.974
1.107	1.016	1.416	0.869	0.048	4.355
0.905	0.883	0.933	0.668	0.049	2.362
1.109	1.083	1.133	0.517	0.047	1.795
1.124	1.083	1.166	0.092	0.049	2.154
1.157	1.116	1.183	0.271	0.048	9.836
1.083	1.050	1.116	0.105	0.048	3.537
1.374	1.216	1.533	3.184	0.048	115.992
1.141	1.100	1.166	0.337	0.049	20.949
1.221	1.100	1.483	1.348	0.049	9.670
1.143	0.883	1.533	0.760	0.047	115.992

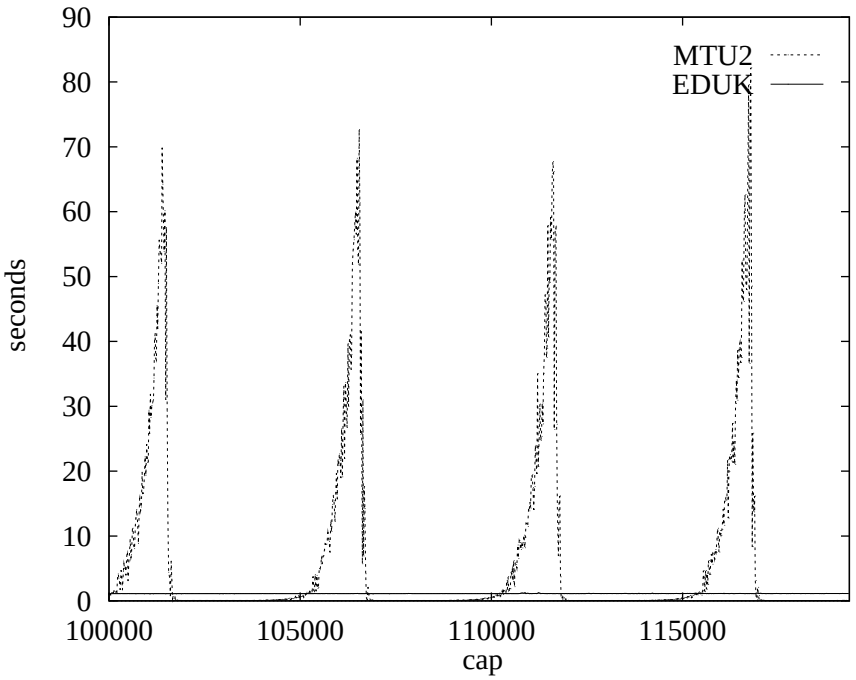
Average, maximum and minimum running times (in seconds) of EDUK and MTU2, for uncorrelated (above) and weakly correlated (below) data sets. Each line corresponds to 500 instances (each for a random capacity in the interval $[10^5, 2.6 \times 10^6]$) for a given set of weights and profits chosen randomly with the following parameters: $m = 5 \times 10^4$, $w_{\max} = 10^5$ and $w_{\min} = 5 \times 10^3$. The last line in each table is the average of the columns.

Figure 6: Capacity sensitivity of EDUK and MTU2

experiment: we set $w_{\min} = 5000$, $w_{\max} = 10^5$ and $m = 50000$. For each set of values of weights and profits, we measured the running time for 500 values of capacity, randomly chosen in the interval $[10^5, 2.6 \times 10^6]$. Fig. 6 presents the average, minimum and maximum of these values for the two algorithms (since MTU2 very often requires more than 1000 seconds for strongly correlated data sets with these parameters, we report here only results for non-correlated and weakly correlated data sets).

These figures illustrate that the time required by EDUK is stable, the average time is at the middle of the interval and the maximal time is two times the minimal one. On the same examples the maximal time required by MTU2 can be 2500 times its minimal time.

In order to try to illustrate the behavior more precisely, we conducted 1000 similar runs for a particular (weakly correlated) data set from the above experiment, but chose the capacity randomly in the range $[10^5, 119400]$. Fig 7 reports the running time of both algorithms for each value of the capacity. We observe that the running time of MTU2 is



Running times for 1000 runs of EDUK and MTU2, for a particular instance of uncorrelated (?) data set, each for a random capacity in the interval $[10^5, 119400]$. The average (max/min) times for EDUK and MTU2 were respectively, 1.119 (1.08/1.25) seconds, and 6.731 (0.028/82.184) seconds.

Figure 7: Capacity Sensitivity of EDUK and MTU2 (cont'd)

nearly periodic with a period equal to the weight of the best object type (here, about 5000) while that of EDUK is constant.

We conclude that EDUK is more stable than MTU2 to variations in the capacity. This is due to the way in which EDUK exploits *threshold dominance* dominance (and hence *periodicity*), and also the *sparse representation*.

4.2.1 Realistic Random Data Sets

In real life, it is almost inconceivable that a heavier object has smaller profit (i.e., there cannot be any simple dominance). ***Realistically random*** data sets are simply those that preclude simple dominance by fiat. They may be generated as follows: first generate m distinct values of weights, randomly in the interval $[w_{\min}, w_{\max}]$, and m values of profits randomly in the interval $[p_{\min}, p_{\max}]$. Then (i) sort these two sequences, (ii) pair up the weights and profits, and (iii) permute this list randomly (or return it to the original order in which (one of) the weights (or profits) were generated). This yields a realistic *uncorrelated*[2]

[2] Actually, the term uncorrelated is a misnomer, since there is a correlation (though not the linear/affine one of the strongly correlated data sets) between the weights and profits: a larger weight implies a greater profit.

$w_{\min}$	UD	eduk	mtu2
1	6	0.098	0.021
3	69	0.873	$1 > 1000$
10	460^*	$1 > 1000$	$3 > 1000$
30	1780	111.075	$3 > 1000$
100	7581^*	$4 > 1000$	$9 > 1000$
300	27724^*	$10 > 1000$	$9 > 1000$
1000	$-^*$	$10 > 1000$	$10 > 1000$
3000	$-^*$	$10 > 1000$	$10 > 1000$

Realistic data sets are those where there is no simple dominance: larger weight implies a larger profit. The parameters were: $m = 5 \times 10^4$, $w_{\max} = 10^5$, $w_i \in [w_{\min}, w_{\max}]$, $p_i \in [w_i - 100, w_i + 100]$ and $c > 2 \times 10^8$. Both EDUK and MTU2 have fairly poor performance on such data sets. Each line is the average of 10 runs on different instances, except for the lines containing an asterix, where we were unable to solve (some instances of) the problem within 3000 sec. In such cases, the number in the *UD* column is the average of the available data.

Figure 8: EDUK and MTU2 running times on realistic random data sets

data set, and although weakly correlated data sets could also be generated, they do not seem very interesting. Of course, strongly correlated data sets already preclude simple dominance (provided duplicates are ruled out).

These results demonstrate that for this family of problems, the UKP is a difficult problem for both EDUK and MTU2 although EDUK remains faster. This is due to the fact that here, the number of undominated object types (for $\ll$) grows rapidly with w_{min} and then the "real" size to be considered, which depends of this number, is actually large. We conjecture that interesting real life problems fall in this family.

4.3 Hard Problems

Since the UKP problem is known to be NP-complete, it is possible to contrive examples of problems that force any given algorithm into exponential running time. Here, we exhibit two kinds of such hard problems. We found the first one by chance when generating random examples. Problems of the second kind are built with formulæ that ensure that no object type is collectively dominated by the others.

4.3.1 A random example

As seen above, the performance of both EDUK and MTU2 appears to be poor for the realistic random data sets, and one of our hard example arose in the context of such experiments. It was generated with $m = 10^5$, $w_{\min} = 30$, $w_{\max} = 10^5$, $p_{max} = 10^6$ $c = 10^6$. It turned out that for this example, there were 22,077 (collectively) non dominated items, and the threshold of the best object type was greater than the capacity (which was 10^6). EDUK took 3,430 seconds to solve it, while MTU2 failed to solve it in 20,000 seconds.

Prob	eduk	mtu2
$\mathcal{F}_1$	4.316	113.734
$\mathcal{F}_2$	4.183	103.137
$\mathcal{F}_3$	3.983	98.432
$\mathcal{F}_4$	3.800	96.533
$\mathcal{F}_5$	3.866	93.107
$\mathcal{F}_6$	3.866	94.688
$\mathcal{F}_7$	3.766	103.925
$\mathcal{F}_8$	3.666	104.185
$\mathcal{F}_9$	3.633	86.464
$\mathcal{F}_{10}$	3.600	92.914
$\mathcal{F}_{11}$	3.616	77.667
$\mathcal{F}_{12}$	3.733	111.204
$\mathcal{F}_{13}$	3.566	83.674
$\mathcal{F}_{14}$	3.666	78.156
$\mathcal{F}_{15}$	3.533	64.176
$\mathcal{F}_{16}$	3.600	55.614
$\mathcal{F}_{17}$	3.633	91.784
$\mathcal{F}_{18}$	3.600	87.066
$\mathcal{F}_{19}$	3.516	83.365
$\mathcal{F}_{20}$	3.516	72.872
$\mathcal{F}_{21}$	3.483	75.390
$\mathcal{F}_{22}$	3.583	100.651

Prob	eduk	mtu2
$\mathcal{G}_1$	1.716	3.355
$\mathcal{G}_2$	4.950	> 20000
$\mathcal{G}_3$	7.816	0.601
$\mathcal{G}_4$	17.016	0.365
$\mathcal{G}_5$	14.816	2.276
$\mathcal{G}_6$	20.983	> 20000
$\mathcal{G}_7$	21.033	> 20000
$\mathcal{G}_8$	26.016	> 20000
$\mathcal{G}_9$	22.350	0.081
$\mathcal{G}_{10}$	37.716	76.941
$\mathcal{G}_{11}$	30.100	2.331
$\mathcal{G}_{12}$	18.833	> 20000
$\mathcal{G}_{13}$	25.133	> 20000
$\mathcal{G}_{14}$	14.233	> 20000
$\mathcal{G}_{15}$	15.983	> 20000
$\mathcal{G}_{16}$	12.683	1.358
$\mathcal{G}_{17}$	13.366	> 20000
$\mathcal{G}_{18}$	19.150	> 20000
$\mathcal{G}_{19}$	11.966	> 20000
$\mathcal{G}_{20}$	13.133	> 20000
$\mathcal{G}_{21}$	20.683	76.413
$\mathcal{G}_{22}$	8.366	77.602

Prob	eduk	mtu2
$\mathcal{C}$	1643.483	> 20000
Difficult	3430.333	> 20000

Figure 9: Hard problems constructed from a particular realistically random data set.

We used this particular problem to construct two families of smaller but still hard problems (each with 1001 object types). Let $\mathcal{C}$ be the set (called the core set) of the 22,077 non dominated object types, sorted by increasing weight and let $\mathcal{C}_i$ denote the i-th element of $\mathcal{C}$. Thus, $u = \mathcal{C}_{22,077}$ is the heaviest non dominated object type. The first family, $\mathcal{F}$, is constructed by simply partitioning $\mathcal{C}$ "by blocks" into subsets of 1000 consecutive object types, and throwing in u, i.e., $\mathcal{F}_i = \{\mathcal{C}_j | (i-1) < \lceil \frac{j}{1000} \rceil \leq i\} \cup \{u\}$, yielding 22 problems. The second one $\mathcal{G}$ is constructed by partitioning "cyclically", i.e., taking every 20-th element of $\mathcal{C}$. Formally, $\mathcal{G}_i = \{\mathcal{C}_j | j \bmod 20 = i\} \cup \{u\}$. Fig. 9 reports the time needed by EDUK and MTU2 to resolve these problems (as before, > 20000 denotes that the time required exceeds 20,000 seconds). The last table gives the times required to solve the core problem, $\mathcal{C}$ itself, and the original problem with 10^5 object types.

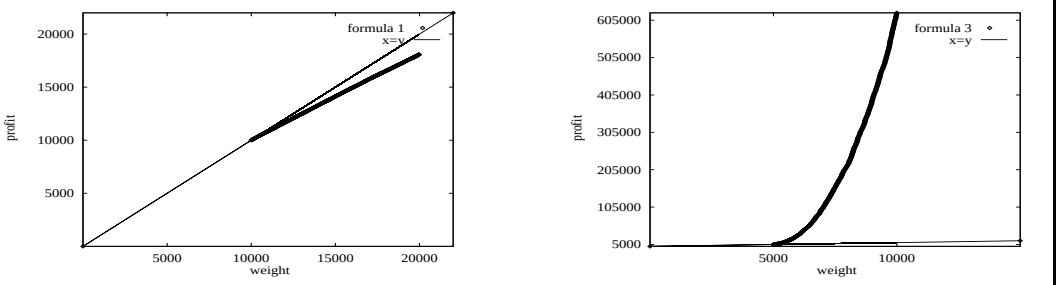


Figure 10: Sample data sets for Eqns. 6 and 7. Observe how the profitability decreases (resp. increases) with weight.

These experiments indicate that EDUK seems to consistently outperform MTU2 on this class of data.

4.3.2 Artificially constructed hard problems

We now construct pathological examples which are hard for the EDUK algorithm. They are based upon the following observations: if the profit monotonically increases with the weight, then there is no simply dominated object type; if, in addition, $w_{\max} < 2w_{\min}$, no object type is maximally dominated. Three such data sets (illustrated in Fig. 10) can be constructed as follows. Choose the weights randomly and sort them in increasing order. The profits are as given by Eqns. (5–7) below (for the latter two, p_1 is chosen randomly).

$$p_i = w_i \tag{5}$$

$$p_i = \max\left(p_{i-1}, \left\lfloor \frac{w_i p_{i-1}}{w_{i-1}} \right\rfloor\right) \tag{6}$$

$$p_i = \left\lfloor \frac{w_i p_{i-1}}{w_{i-1}} \right\rfloor + i - 1 \tag{7}$$

It is easy to verify that in each of the three cases, no object type is collectively dominated, and are distinguished by the fact that the *profitability* of the object types is (i) constant, (ii) decreases or (iii) increases with the weight. The three data sets will have different performance because of threshold dominance. For example, it is well known that if several object types have maximum profitability (as is the case with Eqn. 5) then the periodicity is greater [6]. The weight-profit pairs so generated were randomly permuted before we ran the experiments[3]. Fig. 11 presents the running times which clearly highlight that the problem remains difficult.

[3] As an aside, we observed that MTU2 was very sensitive to this. For example, for a particular data set (with $w_i = p_i$ with 1000 object types and with $w_{\min} = 5000$ and $w_{\max} = 10000$), if the object types are sorted by increasing weight, MTU2 is unable to resolve the problem in 10,000 seconds, while if the same data set is not sorted, it takes 0.007 seconds!

Formula 1			Formula 2			Formula 3		
m	eduk	mtu2	m	eduk	mtu2	m	eduk	mtu2
1000	40.48	> 1000	1000	44.43	0.02	1000	2.78	126.02
2000	116.87	> 1000	2000	63.38	0.01	2000	13.17	> 1000
4000	304.83	> 1000	4000	147.33	0.01	4000	52.97	> 1000
5000	396.55	> 1000	5000	197.87	0.01	5000	88.22	> 1000
10000	> 1000	> 1000	10000	551.73	0.02	10000	403.32	> 1000
20000	> 1000	> 1000	20000	> 1000	0.02	20000	> 1000	> 1000

Figure 11: Hard problems generated by formulæ

4.4 Other Examples

Chung et al. have given a formula to build a family of instances of the UKP [4].

$$w_i = i + w_0; \quad p_i = w_i + z$$

where w_0 and z are parameters. These problems are shown to be *hard* for the branch and bound algorithm family, in the sense that the B&B algorithm takes provably exponential time. Observe that these problem instances are *strongly correlated* in the Martello and Toth sense, except that the weights are generated in order. Chung et al. present a number of instances of the problem, which for even relatively small values of m and c provoke exponential time for branch and bound algorithms (some of them exceed their machine capacity). All the "hard" instances were resolved by EDUK in less than 0.02 seconds[4]!

In practice, one often encounters problems with small capacity (the UKP often arises as a frequently called subproblem in other combinatorial optimization problems). For these problems, it seems that for preprocessing may not be prohibitively expensive, and the overhead of computing by slices may be high. For this reason we wanted to compare the EDUK algorithm with a straightforward dynamic programming algorithm, with a preprocessing step removing object types according to simple dominance, $\ll_s$. We have done that comparison with a set of 265 problems generated with a generator that Valério de Carvalho and Roderiguez [14] have kindly allowed us to use. The values shown below are the average values of: the number of object types, the capacities, the time of EDUK, the time for SDP (standard dynamic programming plus sorting), the time for MTU2 and the number of non-dominated object types (using the two dominance relations).We give also the minimum and maximum time for EDUK and MTU2. They lead us to conclude that the EDUK algorithm is better than the standard DP with preprocessing to remove (simply) dominated object types. Also note that for these example, maximal dominance does not give increased savings (as

[4]We do not claim that this improved performance is due it is due to EDUK. It is well known that even naive dynamic programming gives better results than branch and bound on these examples.

compared to simple dominance) and we conjecture that the improvement comes merely by avoiding the sorting.

m	c	EDUK	MTU2	SDP	$\ll$	$\ll_s$	$eduk_{min}$	$mtu2_{min}$	$eduk_{max}$	$mtu2_{max}$
1288	1035	0.033	0.023	0.039	124.61	125.17	0.	0.	0.066	0.438

5 Conclusions

We have developed some fundamental properties of the unbounded knapsack problem, namely dominance relations, periodicity, sparse representation and efficient backtracking. We incorporated all these properties in the framework of dynamic programming recurrence, using a rigourous transformation program approach (not described here due to space constraints). This allowed us to design an efficient algorithm with very low sensitivity to variations of the knapsack capacity. This stability is one of the most significant differences between dynamic programming and branch and bound algorithms. We illustrate this advantage by comparing our results with MTU2, which is the most widely used today branch and bound algorithm for solving UKP. Our results indicate the viability of dynamic programming for this NP-hard problem.

We validate our ideas by a large number of computational experiments and comparisons. Our results confirm again that even very large randomly generated instances of UKP have few undominated items and can be reduced to small core problems. Therefore the number of object types (m) is not a good measure for the difficulty of the problem. We observe that the value of the minimum weight ($w_{\min}$) is more relevant to indicate the hardness of the instance. However this parameter is not sufficient by itself since independently of its value an UKP is easily solvable when the instance contains simply dominated object items.

We further show that the problem remains time consuming in the case of the so called realistic random instances, i.e., where no object type is simply dominated and $w_{\min}$ is as small as 10.

We have identified non-trivial data instances for UKP by using these criteria in conjunction, no simply dominated object types and increasing $w_{\min}$. We believe that it should be interesting to compare other exact algorithms on these instances, and for this purpose we are developing a database of benchmarks[5].

Acknowledgments The authors are very thankful to Nicola Yanev for many helpful and encouraging discussions, and for his numerous suggestions which contributed significantly for the current version of the paper.

[5]The current version is at `http://www.univ-valenciennes.fr/limav/knapsacks/ukp/datas`

References

[1] R. Andonov, V. Poirriez, and S. Rajopadhye. Efficient Dynamic Programming for the Unbounded Knapsack Problem. Technical Report 96-07, LIMAV, Université de Valenciennes, Le Mont Houy, B.P. 311, 59304 Valenciennes Cedex, France, October 1996.

[2] R. Andonov and S. V. Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *International Conference on Application-Specific Array Processors (ASAP-94)*, pages 302–313, San Francisco, August 1994. IEEE.

[3] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[4] C-S. Chung, M. S. Hung, and W. O. Rom. A hard knapsack problem. *Naval Research Logistics*, 35:85–98, 1988.

[5] K. Dudzinski. A note on dominance relation in unbounded knapsack problem. *Operations Research Letters*, 10:417–419, 1991.

[6] R. Garfinkel and G. Nemhauser. *Integer Programming.* John Wiley and Sons, 1972.

[7] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem – Part II. *Operations Research*, 11:863–888, 1963.

[8] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

[9] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Maryland, USA, 1978.

[10] T. C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, 1969.

[11] R. Johnson and L. Khan. A note on dominance in unbounded knapsack problems. *Asia-Pasific Journal of Operationnal Research*, (12):145–160, 1995.

[12] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[13] D. Pisinger. Dominance Relations in Unbounded Knapsack Problems. Technical Report 94/33, DIKU, University of Copenhagen,DK-2100 Copenhagen,Denmark, December 1994.

[14] J.M. Valério de Carvalho and A.J. Rodrigues. An LP-based Approach to a Two-Stage Cutting Stock Problem. *European Journal of Operational Research*, 84:580–589, 1995.