View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Crossref

International Scholarly Research Network
ISRN Electronics
Volume 2012, Article ID 859820, 10 pages
doi:10.5402/2012/859820

Research Article

Polynomial Time Instances for the IKHO Problem

Romeo Rizzi[1] and Luca Nardin[2]

[1] *Department of Computer Science, University of Verona, 37134 Verona, Italy*
[2] *Department of Information Engineering and Computer Science, University of Trento, 38123 Povo, Italy*

Correspondence should be addressed to Luca Nardin, luk.nardin@gmail.com

Received 20 January 2012; Accepted 7 February 2012

Academic Editors: C. W. Chiou and T. L. Kunii

Copyright © 2012 R. Rizzi and L. Nardin. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Interactive Knapsacks Heuristic Optimization (IKHO) problem is a particular knapsacks model in which, given an array of knapsacks, every insertion in a knapsack affects also the other knapsacks, in terms of weight and profit. The IKHO model was introduced by Isto Aho to model instances of the load clipping problem. The IKHO problem is known to be APX-hard and, motivated by this negative fact, Aho exhibited a few classes of polynomial instances for the IKHO problem. These instances were obtained by limiting the ranges of two structural parameters, c and u, which describe the extent to which an insertion in a knapsack in uences the nearby knapsacks. We identify a new and broad class of instances allowing for a polynomial time algorithm. More precisely, we show that the restriction of IKHO to instances where $(c+2u)/c$ is bounded by a constant can be solved in polynomial time, using dynamic programming.

1. Introduction

Interactive Knapsacks Heuristic Optimization problem (IKHO) is a particular knapsacks model in which, given an array of knapsacks, an insertion in a knapsack influences the nearest knapsacks, in terms both of weight and of profit. It was introduced by Aho in [1], for solving the load clipping problem arising in electricity management application. It belongs to the general framework of the Interactive Knapsacks problems (IK) (also defined in [1]) which has several other applications, for example, in electricity management, single and multiprocessor scheduling, and packing of n-dimensional items to different knapsacks. Since IKHO is NP-complete [1] and APX-hard [2], the research of polynomial time instances is very important. In [3], Aho introduces a few classes of such instances restricting the values of certain parameters of the problem: c and u, which determine the dimension of the influence on other knapsacks caused by an insertion, and K, that limits the number of insertions. We keep on this line of investigation by adding a wide and significant class of polynomial time instances for the IKHO problem in the case when $(c + 2u)/c$ is bounded.

Intuitively, in IKHO, when we insert an item in a knapsack, this item is replicated (*cloned*) to the c next knapsacks (hence forming a *cloning block* over $c + 1$ consecutive knapsacks), and it causes an arbitrary but predetermined modification (*radiation*) of the weight and profit of the knapsacks at distance at most u from the cloning block (on both sides of the cloning block). After a knapsack is involved in a cloning operation, we are not allowed to insert any other item in that knapsack. Therefore, the cloning blocks are disjoint. In this paper, we are mainly interested in the case where the ratio between the whole width $c + 2u$ of the influenced zone (cloning plus radiation zones) and the width c of the cloning part is bounded by a constant r. We propose a dynamic programming algorithm based on a matrix of size $O(c^r \times K)$, and having time complexity of $O(c^r \times m \times K)$, where m is the number of knapsacks and K represents the maximum number of cloning block that we can insert in the knapsacks array.

In Section 2, we give the original formulation of the problem from [1] and we then simplify it to ease our exposition in later sections. In Section 3, we give the algorithm. In Section 4, we sharpen the complexity result.

Finally, in Section 5, we design a memory saving version of that algorithm.

We conclude this section by defining some useful notation. Henceforth, we write $\mathbf{0}^m$ to denote a zero constant vector of length m, that is, $\mathbf{0}_i^m = 0$ for $i = 1, \dots, m$. Moreover, if $x, y \in \{0,1\}^m$, we write $x \cdot y$ to indicate the concatenation of the two binary strings. Furthermore, $[a, b]$ always denotes a range of integers and, if $a > b$, we assume that $[a, b]$ is empty. In the same way, if $a > b$, the notation *for* $i = a, \dots, b$ means *for no* i.

2. Formulation of the IKHO Problem

We are given an array of m knapsacks, each one of capacity b_ℓ, for $\ell = 1, \dots, m$. There is a single item that we are asked to insert at most K times in the knapsacks array, where K is a natural given as part of the input. The profit and weight of an insertion depend on the knapsack in which we insert for $i = 1, \dots, m$ the naturals w_i and p_i represent the weight and the profit of an insertion in the knapsack i. The main feature of IK problems is that every insertion has also an influence on the weight and profit of the nearby knapsacks. In this way, the weight charged on and the profit relative to a knapsack are established by the insertions in all the knapsacks. To describe this mechanism, Aho introduces a function (called interactive function) I_i for each knapsack $i = 1, \dots, m$, that determines the interaction from a knapsack i to every other knapsack. In particular, given naturals c and u, for each knapsack i, we know that

$$
\begin{aligned}
&I_i(\ell) = 1 \quad \text{for knapsacks } \ell \in [i, i+c], \\
&I_i(\ell) \in \mathbb{Q} \quad \text{arbitrary for knapsacks } \ell \in [i-u, i-1] \\
&\qquad\qquad \cup [i+c+1, i+c+u], \\
&I_i(\ell) = 0 \quad \text{for all other knapsacks.}
\end{aligned}
\tag{1}
$$

The range $[i, i+c]$ is called *cloning block* and the range $[i - u, i-1] \cup [i+c+1, i+c+u]$ *radiation part*. The role of the functions I_i gets clear in formulas (2)–(5).

The decision variables used to denote in which knapsacks we do the insertions are $x_i \in \{0,1\}$ for $i = 1, \dots, m$. Given in input $m, p_i, w_i, b_i, c, u \in \mathbb{N}$ and $K \in \mathbb{N} \setminus \{0\}$, an IKHO problem is to

$$\text{maximize} \quad \sum_{i=1}^{m} x_i \sum_{\ell=1}^{m} I_i(\ell) p_\ell \tag{2}$$

$$\text{subject to} \quad \sum_{i=1}^{m} x_i I_i(\ell) w_i \le b_\ell, \quad \text{for } \ell = 1, \dots, m, \tag{3}$$

$$\sum_{i=1}^{m} x_i \le K, \tag{4}$$

$$x_j = 0, \quad \text{for } i < j \le i + c, \text{ when } x_i = 1, \tag{5}$$

where $i = 1, \dots, m$ in (5). Clearly, since a feasible x must belong to $\{0,1\}^m$, then at most one item may be put in each single knapsack. Moreover, notice that in Constraint (3), imposing that the knapsacks are not overfilled, $I_i(\ell)$ is multiplied by the weight w_i. Thus, when we insert in the knapsack $i (x_i = 1)$, the weight w_i is equivalently charged on the knapsacks $\ell \in [i, i+c]$, for which $I_i(\ell) = 1$. This is the reason of calling the range $[i, i+c]$ *cloning block*. Regarding the knapsacks in the radiation part $[i-u, i-1] \cup [i+c+1, i+c+u]$, we get that an arbitrary portion of the weight w_i is added or subtracted (since $I_i(\ell)$ can be negative) to them. Similar operations are performed in the maximization function (2) with the profits p_ℓ. Furthermore, Constraint (4) specifies the maximum number of cloning blocks to be put into the knapsacks array, while Constraint (5) tells that the cloning blocks must be disjoint. The IKHO model is more widely explained and motivated in [1, 4].

2.1. Simplifying the Formulation. Our first step here is to simplify the formulation of the problem by making the notion of weight independent from the interaction functions, and the profit dependent only on the knapsack where we insert. This is accomplished by exploiting the transformation proposed by Aho in [2], in order to reduce IKHO to MDKP, an IPL formulation surveyed in [5]. We define

$$
\begin{aligned}
p_i' &= \sum_{\ell=1}^{m} I_i(\ell) p_\ell, \\
w_{i\ell}' &= I_i(\ell) w_i,
\end{aligned}
\tag{6}
$$

so that p_i' represents the total profit of an insertion in the knapsack i, and $w_{i\ell}'$ is the weight that an insertion in the knapsack i charges over the knapsack ℓ. From the features of $I_i(\ell)$ exposed in (1), it follows that $w_{i\ell}'$ and p_i' are both rational numbers. Thus, notice that they can be also negative.

Now, we can reformulate the problem as follows:

$$\text{maximize} \quad \sum_{i=1}^{m} x_i p_i' \tag{7}$$

$$
\begin{aligned}
&\text{subject to} \quad \sum_{i=1}^{m} x_i w_{i\ell}' \le b_\ell, \quad \text{for } \ell = 1, \dots, m, \\
&\qquad (4) \text{ and } (5).
\end{aligned}
\tag{8}
$$

Henceforth we refer always to the latter formulation of the problem, since it simplifies the description of the algorithm.

Let us restate the behavior of parameter w' as inherited by functions I_i. For $i = 1, \dots, m$, we have that

$$
\begin{aligned}
&w_{i\ell}' = w_i \text{ (and then it is constant)} \quad \text{for } \ell \in [i, i+c], \\
&w_{i\ell}' \text{ is arbitrary} \quad \text{for } \ell \in [i-u, i-1] \cup [i+c+1, i+c+u], \\
&w_{i\ell}' = 0 \quad \text{for all other } \ell \in [1, \dots, m].
\end{aligned}
\tag{9}
$$

2.2. Polynomial Time Instances. The classes of instances isolated by Aho are the following:

(a) the instances with $c = u = 0$;

(b) those with $K = O(1)$;

(c) those with $u = 0$;

(d) those with $c + 2u + 1 = O(\log(m^{\alpha}))$, for a constant α.

The restriction of IKHO obtained by considering only the instances in (a) corresponds to the situation in which there are no interactions, whence the decision on whether to insert an item can be taken independently on each knapsack. As for (b), notice that any instance of IKHO admits at most m^K feasible solutions, which is only a polynomial number of possibilities whenever $K = O(1)$. We refer to [3] for details on Aho's algorithm for instances of type (c) and (d).

In Section 3, we describe an algorithm for IKHO that has time complexity of $O(m \times K \times c^{(c+2u)/c})$. When $(c + 2u)/c$ is bounded by a constant, it clearly becomes a polynomial time algorithm. Indeed, note that the term $c^{(c+2u)/c}$ is polynomial also when $c + 2u = O(\ln(m^{\alpha}))$, for a constant α. In fact, $c^{(c+2u)/c} = (c^{1/c})^{c+2u}$, and $c^{1/c}$ is a decreasing function on c. Therefore, our results imply those reported in (a), (c), and (d).

3. The Algorithm

In the following, $A = (m, b, c, u, w', p', K)$ always denotes the input IKHO instance. Let $L := c + 2u$. A binary string $s \in \{0,1\}^L$ is called a *signature* if it obeys Constraint (5), that is, if $s_i + s_j \le 1$ for all $i, j \in [1, L]$ such that $1 \le |i - j| \le c$. We denote by $\mathcal{S}$ the set of all signatures.

Given a solution $x \in \{0,1\}^m$, $|x| := \sum_{i=1}^{m} x_i$ denotes the number of insertions prescribed by x. Moreover, for each $\ell \in [1, m]$, $w(x, \ell) := \sum_{i=1}^{m} x_i w'_{i\ell}$ is the weight charged on the knapsack ℓ by the solution x. Then, we say that a solution x obeys the *capacity constraint* (Constraint (8)) on the knapsacks $[\alpha, \beta]$ if and only if $w(x, \ell) \le b_\ell$ for each $\ell \in [\alpha, \beta]$. Furthermore, we write $x \vdash (s, h)$ when $x_{h+j} = s_j$ for each $j \in [1, L]$ and $x_i = 0$ for each $i \in [1, h]$, that is, when *x starts with* the signature s in the knapsacks $[h + 1, h + L]$, whence having the form $x = \underbrace{0 \cdots 0}_{h} s x'$.

3.1. The Subproblems of Our DP Approach. Given a natural $k \le K$, a natural $h \le m - L$, and a signature s, we consider a modified problem Sub$[k, h, s]$, whose solutions are those $x \in \{0,1\}^m$ which obey

$$x \vdash (s, h), \tag{10}$$

$$w(x, \ell) \le b_\ell \quad \forall \ell \in [h + c + u + 1, m], \tag{11}$$

$$|x| \le k, \quad \text{and (5).} \tag{12}$$

The objective function is the same as in the IKHO formulation. The differences between the IKHO problem and the above-defined subproblems are in the additional parameters s, h, k and their use in the constraints.

(i) Constraint (10) fixes the first insertions in compliance to the signature s.

(ii) The range on which we check the capacity constraint in (11) is $[h+c+u+1, m]$, a subset of the range $[1, m]$ checked in IKHO.

(iii) By Constraint (12), we can do at most k insertions. Notice that in general (12) is more restrictive than (4).

In the following, we denote by $\mathcal{X}[A]$ the space of solutions to the IKHO instance A, and by $\mathcal{X}[k, h, s]$ the space of solutions to the modified problem Sub$[k, h, s]$. Moreover, let opt$[A]$ be the maximum value of a solution in $\mathcal{X}[A]$ and opt$[k, h, s]$ the maximum value of a solution in $\mathcal{X}[k, h, s]$. It is assumed that opt$[k, h, s] = -\infty$ when $\mathcal{X}[k, h, s]$ is empty.

3.2. The Dynamic Programming Algorithm. Our dynamic programming approach is based on Lemmas 1 and 2, whose proofs are given later in this subsection. In particular, Lemma 1 shows how to read out an optimal solution to the IKHO instance, from the optimal solutions to the subproblems.

Lemma 1. *Let $\mathcal{S}_0 \subseteq \mathcal{S}$ be the set of the signatures s such that $y = s \cdot \mathbf{0}^{m-L}$ obeys the capacity constraint on all knapsacks $\ell \in [1, c + u]$. Then,*

$$\text{opt}[A] = \max_{s \in \mathcal{S}_0} \text{opt}[K, 0, s]. \tag{13}$$

Indeed, $\mathcal{X}[A] = \bigcup_{s \in \mathcal{S}_0} \mathcal{X}[K, 0, s]$.

Lemma 2 explains how to recursively solve the subproblems. We need some additional notation. Given $x \in \{0,1\}^m$, $b \in \{0,1\}$, for $i = 1, \dots, m$, we write $x + (b, i)$ to denote the binary string obtained from x by setting its i-th element to b. Moreover, if $X \subseteq \{0,1\}^m$, we let $X + (b, i) := \{x + (b, i) : x \in X\}$. Furthermore, for each $b \in \{0,1\}$, and $h \in [0, m - L]$, a signature $s \in \mathcal{S}$ is called (h, b)*-good* if $z := \mathbf{0}^h \cdot s \cdot b \cdot \mathbf{0}^{m-h-L-1}$ obeys the capacity constraint for the knapsack $h + c + u + 1$ and if $s' := (s_2, s_3, \dots, s_L, b) \in \mathcal{S}$.

Lemma 2. *For $k = 1, \dots, K$, $h = 0, \dots, m - L - 1$, $s \in \mathcal{S}$,*

$$\begin{aligned} \text{opt}[k, h, s] = \max_{b \in \{0,1\} : s \text{ is } (h,b)\text{-good}} & \text{opt} \\ & \times [k - s_1, h + 1, s' := (s_2, s_3, \dots, s_L, b)] \\ & + s_1 p'_{h+1}. \end{aligned} \tag{14}$$

Indeed,

$$\begin{aligned} &\mathcal{X}[k, h, s] \\ &= \bigcup_{b \in \{0,1\} : s \text{ is } (h,b)\text{-good}} \mathcal{X}[k - s_1, h + 1, s' := (s_2, s_3, \dots, s_L, b)] \\ &\quad + (s_1, h + 1). \end{aligned} \tag{15}$$

The base for the recursion, that is, the cases where $h = m - L$ and $k = 0$, is handled in Section 3.3.

In order to prove Lemmas 1 and 2, let us begin by pointing out some basic facts that directly derive from the IKHO formulation. Observations 1 and 2 play an important

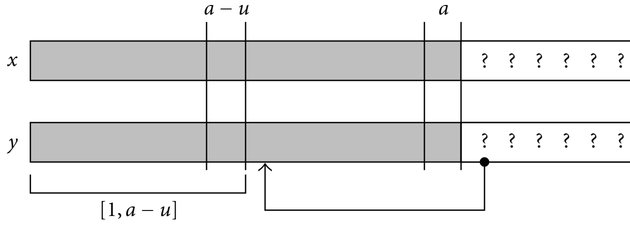


Figure 1: Representing Observation 1. The gray area indicates where x is equal to y, that is, the knapsacks $[1, a]$. The arrow represents the left most possible radiation as starting from the left most possible different bit. Since this radiation cannot reach the knapsacks $[1, a - u]$, then, for $\ell \in [1, a - u]$, $w(x, \ell)$ and $w(y, \ell)$ cannot differ as they depend only on the common bits $[1, a]$.

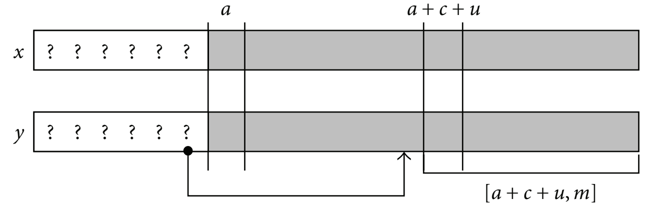


Figure 2: Representing Observation 2.

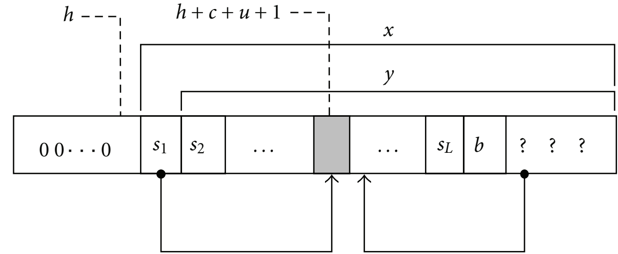


Figure 3: Building a feasible solution to Sub$[k, h, s]$. In order to construct a feasible solution x to the current subproblem Sub$[k, h, s]$, we exploit a feasible solution y to one of the subsequent subproblems Sub$[k-s_1, h+1, s' := (s_2, s_3, \ldots, s_L, b)]$, for a $b \in \{0, 1\}$. Clearly, x must satisfy the capacity constraint on the knapsacks $[h + c + u + 1, m]$. Moreover, notice that the radiation from the bit s_1 does not reach the knapsacks after $h + c + u + 1$, and then, for those knapsacks, x satisfies the capacity constraint only if y does. Therefore, we need to check the capacity constraint only for the knapsack $h+c+u+1$. Furthermore, the bits after b are not involved when we check the capacity constraint on that knapsack. Thus, in order to check the capacity constraint on the knapsack $h + c + u + 1$, it is enough to know the bits of the string $s \cdot b$. Notice that, for a signature s, L is the smallest width that holds the properties we showed above.

role in the formal proofs of these lemmas. For this reason, these observations and their proofs are visualized in Figures 1 and 2, respectively.

Observation 1. Assume $a > u$. Let $x, y \in \{0, 1\}^m$ such that $x_i = y_i$ for $i = 1, \ldots, a$. Then, for each $\ell \le a - u$, x satisfies the capacity constraint if and only if y satisfies it. Indeed, for each $\ell \le a - u$, $w(x, \ell) = w(y, \ell)$.

Proof. Let $\ell \le a - u$. Remember that $w(x, \ell) := \sum_{i=1}^{m} x_i w'_{i\ell}$. However, by (9), $w'_{i\ell} = 0$ for $\ell < i - u$, that is, when $i > l + u$. Moreover, $x_i = y_i$ for $i = 1, \ldots, \ell + u$, since $\ell + u \le a$. Therefore,

$$w(x, \ell) := \sum_{i=1}^{m} x_i w'_{i\ell} = \sum_{i=1}^{\ell+u} x_i w'_{i\ell} = \sum_{i=1}^{\ell+u} y_i w'_{i\ell} \qquad (16)$$
$$= \sum_{i=1}^{m} y_i w'_{i\ell} =: w(y, \ell).$$

□

Observation 2 covers the left/right-reverse situation.

Observation 2. Assume $a \le m - c - u$. Let $x, y \in \{0, 1\}^m$ such that $x_i = y_i$ for each $i = a, \ldots, m$. Then, for each $\ell \ge a + c + u$, x satisfies the capacity constraint if and only if y satisfies it. Indeed, for each $\ell \ge a + c + u$, $w(x, \ell) = w(y, \ell)$.

Proof. Let $\ell \ge a + c + u$. By (9), $w'_{i\ell} = 0$ for $\ell > i + c + u$, that is, when $i < \ell - c - u$. Moreover, $x_i = y_i$ for $i = \ell - c - u, \ldots, m$, since $\ell - c - u \ge a$. Therefore,

$$w(x, \ell) := \sum_{i=1}^{m} x_i w'_{i\ell} = \sum_{i=\ell-c-u}^{m} x_i w'_{i\ell} = \sum_{i=\ell-c-u}^{m} y_i w'_{i\ell} \qquad (17)$$
$$= \sum_{i=1}^{m} y_i w'_{i\ell} =: w(y, \ell).$$

□

Now, we are ready to prove Lemmas 1 and 2.

Proof of Lemma 1. First, let us show that every feasible solution to IKHO is a feasible solution to one of the subproblems Sub$[K, 0, s]$ for an $s \in \mathcal{S}_0$. Clearly, for each $x \in \mathcal{X}[A]$, taking $s := (x_1, x_2, \ldots, x_L)$, we get that $x \in \mathcal{X}[K, 0, s]$. By exploiting Observation 1 with $a = L$, we get that $y := s \cdot \mathbf{0}^{m-L}$ obeys the capacity constraint on knapsacks $[1, c + u]$, and then $s \in \mathcal{S}_0$. To prove the opposite inclusion, take $s \in \mathcal{S}_0$ and $x \in \mathcal{X}[K, 0, s]$. We show that $x \in \mathcal{X}[A]$. Constraint (4), Constraint (5), and the capacity constraint on knapsacks $[c + u + 1, m]$ are clearly satisfied. Since $s \in \mathcal{S}_0$, Observation 1 let us verify the capacity constraint on knapsacks $[1, c + u]$. □

Lemma 2 directly follows from the two opposite inclusions, that we show separately. While reading these proofs, Figure 3 can be useful to visualize the structure of the vectors x, y involved in the proofs.

Proof of Lemma 2. First, we show that

$$\mathcal{X}[k, h, s] \subseteq \bigcup_{b \in \{0,1\} : s \text{ is } (h,b)\text{-good}} \mathcal{X}[k - s_1, h + 1, s' := (s_2, s_3, \ldots, s_L, b)] + (s_1, h + 1). \qquad (18)$$

Proof. Suppose $x \in \mathcal{X}[k, h, s]$. Let $b := x_{h+L+1}$. The inclusion follows by two facts:

(a) s is (h, b)-good;

(b) $x \in \mathcal{X}[k - s_1, h + 1, s'] + (s_1, h + 1)$.

Take $z := \mathbf{0}^h \cdot s \cdot b \cdot \mathbf{0}^{m-h-L-1}$. Since x obeys the capacity constraint on knapsacks $[h+c+u+1, m]$ and $x_i = z_i$ for $i = 1, \ldots, h+L+1$, by exploiting Observation 1 with $a = h+L+1$, we get that z satisfies the capacity constraint on the knapsack $h+c+u+1$. Clearly, $s' := (s_2, s_3, \ldots, s_L, b)$ satisfies Constraint (5), being a substring of x. Hence, s is (h, b)-good.

In order to show that $x \in \mathcal{X}[k-s_1, h+1, s'] + (s_1, h+1)$, we take $y := x + (0, h+1)$ and we show that $y \in \mathcal{X}[k-s_1, h+1, s']$. By following the subproblems definition, it is simple to verify that y obeys Constraint (10), (12), and (5) of $\text{Sub}[k-s_1, h+1, s']$. Moreover, since x satisfies the capacity constraint for each $\ell \in [h+c+u+1, m]$, by applying Observation 2 with $a = h+2$, we get that y satisfies the capacity constraint over the knapsacks $[h+c+u+2, m]$, and then y obeys Constraint (11) too. □

Second, we prove that

$$\bigcup_{b \in \{0,1\}: s \text{ is } (h,b)\text{-good}} \mathcal{X}[k - s_1, h+1, s' := (s_2, s_3, \ldots, s_L, b)] + (s_1, h+1) \subseteq \mathcal{X}[k, h, s]. \quad (19)$$

Proof. Take a $b \in \{0,1\}$ such that s is (h, b)-good and a $y \in \mathcal{X}[k - s_1, h+1, s' := s_2, s_3, \ldots, s_L, b]$. We will show that $x := y + (s_1, h+1) \in \mathcal{X}[k, h, s]$. Constraint (10) and (12) of $\text{Sub}[k, h, s]$ easily follow from subproblems definition. Moreover, since y satisfies Constraint (5) and $s \in \mathcal{S}$, then x satisfies Constraint (5).

It remains to verify Constraint (11). Since s is (h, b)-good, then $z := \mathbf{0}^h \cdot s \cdot b \cdot \mathbf{0}^{m-h-L-1}$ obeys the capacity constraint on the knapsack $h+c+u+1$. By applying Observation 1 with $a = h+L+1$, we derive that also x obeys the capacity constraint on that knapsack. Moreover, since $y \in \mathcal{X}[k - s_1, h+1, s']$, then y obeys the capacity constraint on the knapsacks $[h+c+u+2, m]$. Since $y_i = x_i$ for $i = (h+2, \ldots, m)$, by Observation 2, x obeys the capacity constraint also for the knapsacks $[h+c+u+2, m]$. □

3.3. The Base of the Recursion. We have two base cases. Observation 3 handles the case when $h = m - L$, while Observation 4 treats the case when $k = 0$.

Observation 3. Consider $h = m - L$, for all $k = 1, \ldots, K$, and $s \in \mathcal{S}$. Moreover, let $z := \mathbf{0}^h \cdot s$.

If $|z| \leq k$ and $w(z, \ell) \leq b_\ell$ for each $\ell \in [m-u+1, m]$, then $\text{opt}[k, h, s] = \sum_{i=1}^{m} z_i p'_i$. Otherwise, $\text{opt}[k, h, s] = -\infty$.

Proof. Clearly, since $h + L = m$ and by Constraint (10), there cannot exist a solution to $\text{Sub}[k, h, s]$ different from z. Moreover, notice that $h+c+u+1 = m-u+1$. □

Observation 4. Consider $k = 0$, for all $h = 0, \ldots, m - L$, and $s \in \mathcal{S}$.

If $s = \mathbf{0}^L$, then $\text{opt}[k, h, s] = 0$,

Otherwise $\text{opt}[k, h, s] = -\infty$.

Proof. Clearly, by Constraint (12), $\mathbf{0}^m$ can be the only solution to $\text{Sub}[0, h, s]$. □

4. Complexity

In this section, we prove Lemma 3.

Lemma 3. *Let r be a constant such that L/c is bounded by r when $c > 0$, and L is bounded by r when $c = 0$. The above algorithm takes time and space of $O(c^r \times m \times K)$.*

Clearly, our algorithm exploits a three-dimensional matrix for storing the values $\text{opt}[k, h, s]$, for $k = 1, \ldots, K$, $h = 0, \ldots, m - L$, and $s \in \mathcal{S}$. We also need a matrix of the same size which traces, for each subproblem $\text{Sub}[k, h, s]$, the subsequent subproblem used to compute $\text{opt}[k, h, s]$. This makes us rebuild the optimum solution at the end. The space complexity of the algorithm then is $O(K \times m \times |\mathcal{S}|)$. We need to rate the value of $|\mathcal{S}|$, but first let us to compute the time complexity.

In order to evaluate the base case of our dynamic programming algorithm, we first refer to Observation 3. Clearly, for $k < |s|$, $\text{opt}[k, h, s] = -\infty$, because there must be at least $|s|$ insertion in a solution that starts with the signature s. Moreover, since $z := \mathbf{0}^h \cdot s$ is the only feasible solution to $\text{Sub}[k, h, s]$, it is clear that, for each $k > |s|$, $\text{opt}[k, h, s] = \text{opt}[|s|, h, s]$, by Constraint (12). Therefore, we have to compute only for $k = |s|$, and then, the number of base case subproblems to be computed is only $|\mathcal{S}|$. Since $s \in \{0,1\}^L$, L is the time needed for computing both the profit of z and $w(z, \ell)$, for an $\ell \in [m - u + 1, m]$. Then, to check the capacity constraint on all the knapsacks $[m - u + 1, m]$, we need $u \times L$ computations. Thus, the base case $h = m - L$ can be computed in time of $O(|\mathcal{S}| \times L \times u)$. About the base case $k = 0$, as handled by Observation 4, note that if we codify the signatures $s \in \mathcal{S}$ (a such codify is given in Section 4.2), we can check the condition $s = \mathbf{0}^L$ in $O(1)$. Regarding the general case, by (14), for solving a subproblem, we have to check if s is (h, b)-good, for $b = 0, 1$. To check if $z := \mathbf{0}^h \cdot s \cdot b \cdot \mathbf{0}^{m-h-L-1}$ satisfies the capacity constraint on the knapsack $h + c + u + 1$, and if $s' \in \mathcal{S}$, we spend $O(L)$ computations. Since the number of subproblems is $K \times (m - L) \times |\mathcal{S}|$, we need $O(|\mathcal{S}| \times m \times L \times K)$ time to fill the matrix opt. Moreover, by (13), we have to scan over the $s \in \mathcal{S}_0$ in order to find the maximum value of $\text{opt}[K, 0, s]$. Clearly, $|\mathcal{S}_0| \leq |\mathcal{S}|$, but we need $L \times (c + u)$ computations to check the capacity constraints on all knapsacks $\ell \in [1, c + u]$, because each signature s has width L. Thus, we spend $O(|\mathcal{S}| \times L^2)$ computations, to find $\text{opt}[A]$. Furthermore, rebuilding the best solution takes $O(m)$ time. Therefore, the part in which we recursively compute the subproblems leads the complexity of the entire algorithm. It depends on the value of $|\mathcal{S}|$, as well as the space complexity. In Section 4.1, we give an estimate of this value. Moreover, in Section 4.2, we show an ordering of the set $\mathcal{S}$, that permits us to check the capacity constraint on a knapsack in constant time.

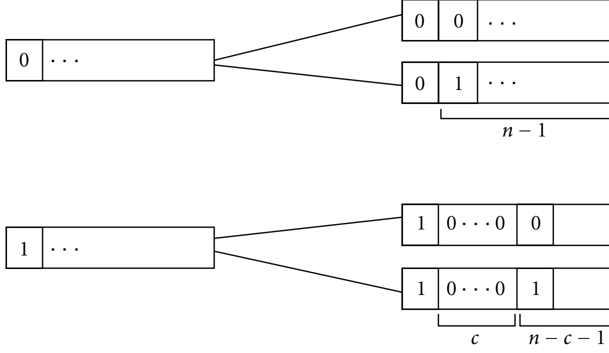


Figure 4: The influence of the first bit choice.

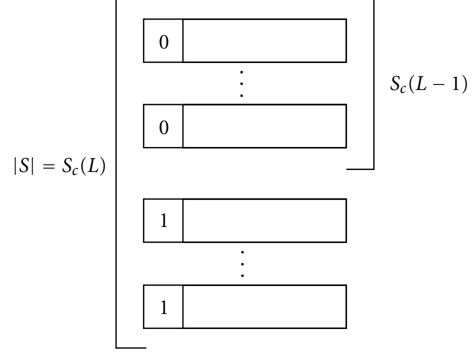


Figure 5: Ordering the signatures.

4.1. Estimating $|\mathcal{S}|$. In the case where c is a constant, we can directly estimate $|\mathcal{S}|$.

Observation 5. If c is a constant, then $|\mathcal{S}| = O(1)$.

Proof. Clearly, $|\mathcal{S}| \leq 2^L$, because the number of binary strings $s \in \{0,1\}^L$ is 2^L. Moreover, when $c = 0$, we supposed L constant. When $c > 0$, since $L/c \leq r$, we get that $L \leq rc$, with constant r and c. □

For nonconstant values of c, let us find a general form for $|\mathcal{S}|$. Let $S_c(n)$ denote the number of binary strings $s \in \{0,1\}^n$ such that s obeys Constraint (5). Note that Constraint (5) contains the parameter c. When $n \in [0, c]$, we have n places where to insert, and at most one insertion is possible by Constraint (5). Moreover, we have to count the string with no insertions. Thus, we get that $S_c(n) = n + 1$ for all $n \in [0, c]$. For greater values of n, we refer to the recursion shown in Figure 4. If the first bit of s is 0, the choice of the following bits is not influenced. Therefore, it is enough to find the number of strings $s \in \{0,1\}^{n-1}$ such that s obeys Constraint (5), that is exactly $S_c(n-1)$. If the first bit is 1, by Constraint (5), the following c bits are necessarily 0's. In this case, we continue to choose after the $c+1$-th bit. Thus, we have $S_c(n-c-1)$ ways to choose the remaining bits. Therefore, we can express $S_c(n)$ by the recurrence equation:

$$\begin{aligned} S_c(n) &= S_c(n-1) + S_c(n-c-1), \\ S_c(n) &= n+1 \qquad \forall n \in [0, c]. \end{aligned} \tag{20}$$

Lemma 4 gives a general estimate of the recurrence $S_c(n)$, in order to bound $|\mathcal{S}|$ for nonconstant values of c.

Lemma 4. *Let $c \geq e$. For each $n \geq c$, $S_c(n) \leq ((c+1)/c)\ c^{n/c}$.*

Proof. We prove the claim by induction on n, and we postpone to the appendix the proof of the base of the induction, that is, the case $c \leq n \leq 2c$. For $n > 2c$, we have the step of induction. Clearly,

$$\begin{aligned} S_c(n) &= S_c(n-1) + S_c(n-c-1) \\ &\leq \frac{c+1}{c}\ c^{(n-1)/c} + \frac{c+1}{c}\ c^{(n-c-1)/c} \\ &= \frac{c+1}{c}\ \left(c^{(n-1)/c} + c^{(n-c-1)/c}\right) \\ &= \frac{c+1}{c}\ c^{(n-c-1)/c}\left(1 + c^{c/c}\right) \\ &= \frac{c+1}{c}\ c^{(n-c-1)/c}(1+c) \\ &= \frac{c+1}{c}\ c^{n/c}\frac{(1+c)}{c^{(c+1)/c}}. \end{aligned} \tag{21}$$

Hence, it is sufficient to show that $(1+c)/c^{(c+1)/c} \leq 1$ or equivalently $1 + c \leq c^{(c+1)/c}$. Since $c^{(c+1)/c} = c \cdot c^{1/c}$, it remains to show that $(1 + 1/c) \leq c^{1/c}$.

We know that $e^x \geq x + 1$ for each real x. By substituting x with $1/c$ and noticing that $c \geq e$, we get that $(1 + 1/c) \leq c^{1/c}$. □

Since $L = c + 2u \geq c$, we can apply Lemma 4 to deduce that $|\mathcal{S}| = S_c(L) = O(c^{L/c})$, as $(c+1)/c \leq 2$ for $c > 0$. Therefore, when L/c is bounded by a constant r, we get $|\mathcal{S}| = O(c^r)$.

4.2. Ranking the Set $\mathcal{S}$. We use Recurrence (20) to define a function $\text{pos} : \mathcal{S} \to [0, |\mathcal{S}| - 1]$, which provides a unique index for each signature, and hence, it gives a ranking for the set $\mathcal{S}$.

Definition 5. For each $s \in \mathcal{S}$,

$$\text{pos}(s) := \sum_{i=1}^{L} s_i \cdot S_c(L-i). \tag{22}$$

Note that $S_c(L-i)$ is the number of signatures having length $L-i$, and it is equivalent to the number of signatures of length $L-i+1$ that start with a 0. Hence, as illustrated in Figure 5, in the step of the sum where $i = 1$, we intend to place the signatures with $s_1 = 1$ after all the signatures with $s_1 = 0$, that are exactly $S_c(L-1)$. For $i = 2, \ldots, L$, we do recursively the same, locating substrings of length $L-i$.

Conversely, given an integer $p \in [0, |S| - 1]$, the unranking procedure is the following. Take $s := \mathbf{0}^L$. For $i = 1, \ldots, L$, if $p \geq S_c(L-i)$, set $s_i = 1$ and $p := p - S_c(L-i)$.

Evidently, in order to efficiently perform such ordering on the set $\mathcal{S}$, we need to compute and store the recurrence $S_c(n)$, for $n = 1, \ldots, L-1$, at the beginning of the algorithm.

This takes $O(L)$ time and space, whereas the ranking and unranking operations take $O(L)$ time.

Indeed, we can avoid to encode and decode the signatures for the computation of each subproblem. This can be done by initializing a table at the beginning of the algorithm, that stores, for each position $\mathrm{pos}(s')$ relative to a signature s', a list of the bits that are changed from the previous signature s, that is, the signature having $\mathrm{pos}(s) = \mathrm{pos}(s') - 1$. It is easy to verify the following procedure finds the next signature from the previous one (it works similarly to the function that increments a binary counter, but considering Constraint (5)).

(i) Scan the previous string s starting from the least significant bit (right most) and find the first range of $c+1$ consecutive 0's, or a range of consecutive 0's that includes the most significant bit (left most).

(ii) If such a range exists, the next string is obtained by setting to 1 the right most bit of the range, and by setting to 0 all the bits at the right of the range.

(iii) If such a range does not exist, s is the last signature.

Above all, observe that, by Constraint (5), there are at most $L/(c+1)$ insertions in a signature $s \in |\mathcal{S}|$, and we know that $L/(c+1) \le L/c \le r$. Therefore, for every kind of ranking for the set $\mathcal{S}$, the number of the bits changing between two adjacent signatures is $O(r)$. Thus, if we know the changing bits from a signature s to the next s', we can use an incremental approach for computing the value $w(s', \ell)$, from $w(s, \ell)$, in constant time. This allows us to check in constant time the capacity constraints of the (h, b)-goodness (14), those regarding the definition of the set $\mathcal{S}_0$ (13), and when computing the base case (Observation 3). Note that also Constraint (5) of the (h, b)-goodness can be computed in $O(r)$ with the same technique.

Moreover, note that when computing the matrix opt, if we place the cycle on the variable s, externally to the cycle on the variable k, we can clearly find the next signature and check the capacity constraints only once every K subproblems. In this way, the cost of finding the next signature is made inessential.

Thus, we can conclude that our algorithm has time and space complexity of $O(c^r \times m \times K)$.

5. Memory Saving Version

Consider (14). Given an $\overline{h} \in [1, m-L-1]$, in order to compute $\mathrm{opt}[k, \overline{h}, s]$ for each $s \in \mathcal{S}$ and $k \le K$, we need only the elements having $h = \overline{h} + 1$ of the matrix opt. Moreover, by (13), only the elements with $h = 0$ are required for computing $\mathrm{opt}[A]$. Thus, in order to work out the profit of a best solution, we only need $O(|\mathcal{S}| \times K)$ space. Unfortunately, this simplification does not apply to the matrix used to compute the best solution.

In [6], Hirschberg showed an elegant and practical space reduction method for the longest common subsequence problem, which works for many dynamic programming algorithms (well exposed also in [7]). In general, this method allows to compute an optimal solution, taking as much space and time as if we had only to compute the optimal solution value. This is accomplished by exploiting the equation which handles the recursion in the original algorithm (in our case (14)). Its space policy exploits the space improvement mentioned at the beginning of this section.

Conceptually, the basic idea of the method is to halve a dimension of the dynamic programming matrix and find how the best solution is divided in the other dimensions. This permits the two halves obtained to be solved separately and recursively in the same way. In order to apply this method to our algorithm, we follow the next steps.

(i) We halve the knapsack array.

(ii) We find how many insertions of the best solution are placed in each half of the knapsack array.

(iii) We locate a number of insertions placed around the middle of the knapsack array. This allows us to break up the IKHO problem in two independent subproblems, which are then solved recursively.

Notice that, in the last sentence, the word *subproblems* does not refer to the subproblems $\mathrm{Sub}[k, h, s]$ defined in Section 3.

In Section 5.1, we implement this idea. In Section 5.2, we show that the new defined algorithm decreases the space complexity to $O(|\mathcal{S}| \times K)$, without increasing the time complexity.

5.1. The Algorithm. In the following, we write $x \vdash^R (s, h)$ when $x_{h-L-1+j} = s_j$ for each $j \in [1, L]$ and $x_i = 0$ for each $i \in [h, m]$, that is, when x *ends with* the signature s in the knapsacks $[h-L, h-1]$.

Given a natural $k \le K$, a natural $h \in [L+1, m+1]$, and a signature s, we consider a modified problem $\mathrm{Sub}^R[k, h, s]$ whose solutions are those $x \in \{0,1\}^m$ which obey

$$\begin{gathered} x \vdash^R (s, h), \\ w(x, \ell) \le b_\ell \quad \forall \ell \in [1, h-u-1], \\ |x| \le k, \\ \text{and } (5). \end{gathered} \tag{23}$$

The subproblems $\mathrm{Sub}^R[k, h, s]$ are simply the symmetrical transposition of the subproblems $\mathrm{Sub}[k, h, s]$. The only significant difference is that we check the capacity constraints in the range $[1, h-u-1]$, that is, not symmetrical to the range $[h+c+u+1, m]$. This dissimilarity is caused by the fact that also the radiations produced by an insertion are not symmetrical too (see (9)).

We denote by $\mathcal{X}^R[k, h, s]$ the space of the feasible solutions to the modified problem $\mathrm{Sub}^R[k, h, s]$. Moreover, $\mathrm{opt}^R[k, h, s]$ is the maximum profit of a solution in $\mathcal{X}^R[k, h, s]$ when $\mathcal{X}^R[k, h, s]$ is not empty, and $\mathrm{opt}^R[k, h, s] = -\infty$ when $\mathcal{X}^R[k, h, s]$ is empty. Notice that, since the subproblems Sub and Sub^R are symmetrical, the properties proved for opt hold symmetrically for opt^R, bringing some adjustments due to the fact that the radiations are not exactly symmetrical. Thus, for computing the matrix opt^R, we take the same space and time needed for computing the matrix opt, that is, $O(|\mathcal{S}| \times K)$ space and $O(m \times |\mathcal{S}| \times K)$ time.

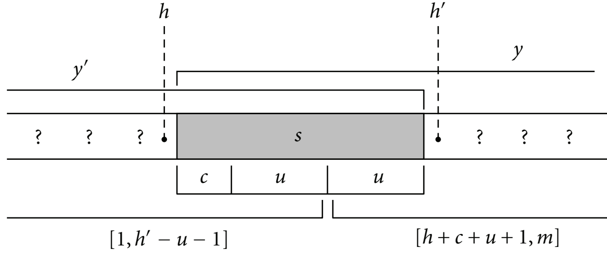


Figure 6: Joining $\mathcal{X}^R[k', h', s]$ with $\mathcal{X}[k, h, s]$. The solution y belongs to the space $\mathcal{X}[k, h, s]$, while $y' \in \mathcal{X}^R[k', h', s]$. Note that the knapsacks $[h + c + u + 1, m]$, where y satisfies the capacity constraint, are complementary to the knapsacks $[1, h' - u - 1]$, where y' satisfies that constraint.

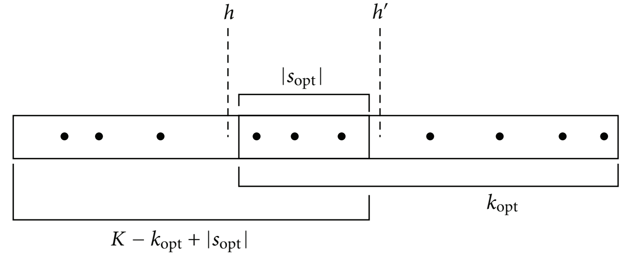


Figure 7: The subdivision on the k dimension. A possible best solution x_{opt} is drawn. The black spots represent the insertions. It is simple to see that the value $k_{\text{opt}} - |s_{\text{opt}}|$ is a bound for the number of insertions placed at the right of s_{opt}, while $K - k_{\text{opt}}$ bounds the number of insertions at the left of the signature.

In the following, let $|s|$ to be the number of insertions in a signature s. For each $s \in \mathcal{S}$, and for $h = 0, \ldots, m - L$, let $p(s, h) := \sum_{i=1}^{L} s_i p'_{h+i}$. Clearly, this function holds for each knapsack h the profit caused by placing the signature s in the knapsacks $[h + 1, \ldots, h + L]$. Moreover, for $\alpha, \beta = 1, \ldots, m$, and $y \in \{0, 1\}^m$, let $y(\alpha, \beta)$ be the substring of y composed by the elements in the range $[\alpha, \beta]$ (we assume $y(\alpha, \beta)$ is empty when $\alpha < \beta$). Furthermore, for $k, k' \leq K$, for each signature s, for $h = 0, \ldots, m - L$ and $h' = L + 1, \ldots, m - L$ such that $h' - h = L + 1$, let $\mathcal{X}^R[k', h', s] \otimes \mathcal{X}[k, h, s] := \{y'(1, h) \cdot s \cdot y(h', m) : y' \in \mathcal{X}^R[k', h', s], y \in \mathcal{X}[k, h, s]\}$. This new operator defines a new space of solutions given by the concatenation of the feasible solutions of two symmetrical subproblems. Notice that the signature s represents the joining point when concatenating the two strings. This situation is represented in Figure 6, which is also useful to visualize the proof of Lemma 6, which represents the main innovation on our algorithm.

Lemma 6. *Let $h = m/2 - L/2 - 1$, and $h' = h + L + 1$. Then,*

$$\text{opt}[A] = \max_{s \in \mathcal{S}, |s| \leq k \leq K} \left(\text{opt}^R[K - k + |s|, h', s] + \text{opt}[k, h, s] - p(s, h) \right). \tag{24}$$

Indeed,

$$\mathcal{X}[A] = \bigcup_{s \in \mathcal{S}, |s| \leq k \leq K} \left(\mathcal{X}^R[K - k + |s|, h', s] \otimes \mathcal{X}[k, h, s] \right). \tag{25}$$

Proof. In order to show that $\mathcal{X}[A] \subseteq \bigcup_{s \in \mathcal{S}, |s| \leq k \leq K} (\mathcal{X}^R[K - k + |s|, h', s] \otimes \mathcal{X}[k, h, s])$, take $x \in \mathcal{X}[A]$. Moreover, take $s := x(h + 1, h' - 1)$, $y := \mathbf{0}^h \cdot s \cdot x(h', m)$, $y' := x(1, h) \cdot s \cdot \mathbf{0}^{m-h'+1}$, and $k := |y|$. In the following, we prove that $y' \in \mathcal{X}^R[K - k + |s|, h', s]$ and $y \in \mathcal{X}[k, h, s]$. Obviously, $y \vdash (s, h)$ and $y' \vdash^R (s, h')$. For which concerning the number of insertions, it is clear that y satisfies Constraint (12) of Sub$[k, h, s]$, as $k = |y|$. Moreover, note that $|x| \leq K$, because $x \in \mathcal{X}[A]$, and $|y'| = |x| - |y| + |s|$. Thus, $|y'| \leq K - k + |s|$, and then y' satisfies Constraint (12) of Sub$[K + |s| - k, h', s]$. About the

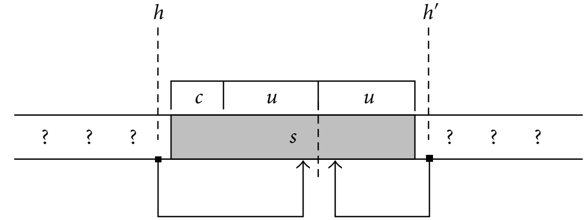


Figure 8: The subdivision on the s dimension. The radiations of weight coming from the right knapsacks do not reach those starting from the left knapsacks.

capacity constraints, in order to show that y satisfies them on the knapsacks $[h + c + u + 1, m]$, we can apply Observation 2 with $a = h + 1$. Conversely, applying Observation 1 with $a = h' - 1$, we obtain that y' satisfies the capacity constraint on the knapsacks $[1, h' - u - 1]$.

To prove the converse inclusion, take $s \in \mathcal{S}$, k such that $|s| \leq k \leq K$, $y' \in \mathcal{X}^R[K - k + |s|, h', s]$, and $y \in \mathcal{X}[k, h, s]$. We prove that $x := y'(1, h) \cdot s \cdot y(h', s) \in \mathcal{X}[A]$. Clearly, since $|y'| \leq K - k + |s|$ and $|y| \leq k$, we get that $|x| = |y'| + |y| - |s| \leq K$, thus x satisfies Constraint (12). Moreover, since $L \geq c$, it is simple to verify that x satisfies also Constraint (5). Furthermore, exploiting Observation 2 with $a = h + 1$, and Observation 1 with $a = h' - 1$, we get that x satisfies the capacity constraint on all the knapsacks $[1, m]$. □

Let s_{opt} and k_{opt} be the values of s and k that maximize (24). Moreover, let $x_{\text{opt}} \in \{0, 1\}^m$ be a best solution for an IKHO instance A. Clearly, the signature s_{opt} represents a piece of x_{opt}, that is, $s_{\text{opt}} = x_{\text{opt}}(h + 1, h' - 1)$. In addition, note that k_{opt} determines a distribution of the best solution insertions, that is, $|x_{\text{opt}}(h+1, m)| \leq k_{\text{opt}}$ and $|x_{\text{opt}}(1, h'-1)| \leq K - k_{\text{opt}} + |s_{\text{opt}}|$. If we do not consider the insertions given by the signature s_{opt}, we obtain $|x_{\text{opt}}(h', m)| \leq k_{\text{opt}} - |s_{\text{opt}}|$ and $|x_{\text{opt}}(1, h)| \leq K - k_{\text{opt}}$, as described in Figure 7.

Having fixed the L middle elements of x_{opt} has an important consequence. The radiations of weight starting from the insertions at the left of s_{opt} cannot interfere with the radiations coming from the insertions placed at the right of s_{opt}, as shown in Figure 8. In particular, by (9), the right

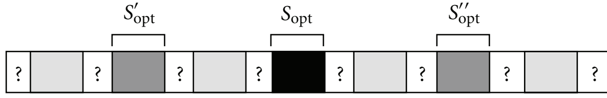


Figure 9: Covering x_{opt} recursively. An optimal solution x_{opt} is drawn. The black elements are determined by the first call of (24). The more gray knapsacks are determined by the two next levels of calls.

insertions affect only the range $[h' - u, m]$, while the left insertions interest only the knapsacks $[1, h + c + u]$.

Therefore, in order to compute the entire part of x_{opt} which stands at the right of s_{opt}, it is enough to know s_{opt}, because the checks on the capacity constraints are independent from the insertions at the left of s_{opt}. The same clearly holds also for finding the left piece of x_{opt}. Thus, we subdivided the main problem in two independent subproblems, as follows:

(i) to find the best solution in the knapsacks $[h', m]$, obeying the capacity constraints over the knapsacks $[h' - u, m]$, having at most $k_{opt} - |s_{opt}|$ insertions and knowing that $x_{opt}(h + 1, h' - 1) = s_{opt}$;

(ii) to find the best solution in the knapsacks $[1, h]$, obeying the capacity constraints over the knapsacks $[1, h + c + u]$, having at most $K - k_{opt}$ insertions and knowing that $x_{opt}(h + 1, h' - 1) = s_{opt}$.

The above-obtained subproblems are solvable as an IKHO instance, by recursively applying (24) with some adjustments. The only significant difference between the main call and the recursive calls is that, in the second ones, when checking the capacity constraints, we have to consider the insertions given by the previously fixed s_{opt}. Indeed, we can simplify this task by progressively updating the vector of capacities b, by subtracting $w(s_{opt}, \ell)$ for each $\ell \in [1, m]$. For each step of recursion, the new vector b will be passed to the input of the lower level subproblems. The solution of each subproblem fixes over x_{opt} a signature s_{opt}, so as to recursively cover x_{opt} with substrings of length L. This is visualized in Figure 9. Finally, note that the base cases for this recursion are the subproblems which involve a number of knapsacks lower than L. In order to compute the best solution for them, we simply list all the feasible solutions and compare the profits.

5.2. Complexity. As mentioned at the beginning of this section, in the first call (root node of the recursion tree), the computing of (24) takes $O(|\mathcal{S}| \times K)$ space. The recursive calls occupy geometrically less memory than the first call, as they deal with an halvened number of knapsacks (and with a lower value of K), whence the total memory consumption is of the same order as the memory consumption for the sole first call. Note that, for each step of recursion, once we found k_{opt} and s_{opt}, we can deallocate the matrices opt and opt^R. Besides these matrices, we need $O(m)$ space for dynamically compose the best solution x_{opt}. Therefore, the new algorithm takes only $O(m + [|\mathcal{S}| \times K])$ space.

Let us now analyze its time complexity. In the first call, for the computation of (24), the algorithm spends $O(m \times |\mathcal{S}| \times K)$ time to compute the matrices opt and opt^R, and $O(|\mathcal{S}| \times K)$ time to pick out the values s_{opt} and k_{opt}. It also spends $O(L^2)$ time in order to update the vector b by subtracting the radiations of weight given by s_{opt}. In fact, a signature has width L, and the range of influence of an insertion is $c + 2u + 1 = L + 1$. To give an estimate for the subproblems computation, we need to study how the two terms found, $O(L^2)$ and $O(m \times |\mathcal{S}| \times K)$, propagate on the next levels of calls. Moreover, let us write them as $\alpha \cdot L^2$ and $\beta \cdot (m \times |\mathcal{S}| \times K)$, for some constants α, β.

First, note that the recursive calling scheme for (24) can be approximated with a binary tree of height $\lceil \log_2(m) \rceil$. Indeed, since the calls on an input of L knapsacks are treated as leaf cases, the height of the binary tree corresponds to the first integer n such that $L \cdot 2^n \geq m$, that is, $\lceil \log_2(m/L) \rceil$. Moreover, it is simple to find that such a binary tree has $O(m/L)$ nodes. In fact, since for each level i, we have at most 2^i nodes, the total number of nodes is

$$\sum_{i=0}^{\log_2(m/L)} 2^i = \frac{1 - 2^{\log_2(m/L)+1}}{1 - 2} = \frac{1 - 2 \cdot (m/L)}{-1} = 2 \cdot \frac{m}{L} - 1. \tag{26}$$

Therefore, considering all the calls of (24), for the first term, we obtain $\alpha L^2 \cdot \gamma(m/L) = O(m \times L)$.

Note that, when we call the two children of a node, both the knapsack array dimension and the K dimension are subdivided. In particular, we call the children resolution on two half of the knapsacks array ($m/2$ and $m/2$), and taking parameters K' and K'' such that $K' + K'' \leq K$. Thus, for each level i, and each node j, the second term of complexity is $\beta \times (m/2^i) \times K_j \times |\mathcal{S}|$, such that $\sum_j K_j \leq K$. Therefore, for the second term, the complexity for each entire level i, is clearly $\beta \times (m/2^i) \times K \times |\mathcal{S}|$. Moreover, by adding up each level, we get that the total complexity for the second term is

$$\begin{aligned}\sum_{i=0}^{\log_2(m/L)} \left(\beta \times \frac{m}{2^i} \times K \times |\mathcal{S}|\right) &= \beta m K |\mathcal{S}| \times \sum_{i=0}^{\log_2(m/L)} \frac{1}{2^i} \\ &\leq \beta m K |\mathcal{S}| \times 2 \\ &= O(mK|\mathcal{S}|).\end{aligned} \tag{27}$$

Since $|\mathcal{S}| = S_c(L) \geq L$, the second term $O(m \times |\mathcal{S}| \times K)$ always bounds the first $O(m \times L)$. Therefore, the complexity of the nodes computation is $O(m \times |\mathcal{S}| \times K)$.

Regarding the computation of the leaf cases, note that we have to find the best feasible piece of solution, for an input of at most L knapsacks. Since these pieces have length lower than L, and they must obey Constraint (5), they are at most $|\mathcal{S}|$. Moreover, we have to consider the cost of checking the capacity constraints on $O(L)$ knapsacks. Thus, a leaf case can be solved in $O(|\mathcal{S}| \times L)$ time. Since we have $O(m/L)$ leaf cases, the total cost of the leaf cases is $O(m \times |\mathcal{S}|)$, that is not higher than the complexity needed for computing the nodes. Therefore, the memory-saving version of the algorithm has time complexity of $O(m \times |\mathcal{S}| \times K)$, which is the same as for the base version.

6. Conclusions

In [3], Aho exhibited a few classes of polynomial instances for the IKHO problem, motivated by the fact that IKHO is NP-complete [1] and APX-hard [2]. The most important class of instances identified by Aho is represented by the instances where $c + 2u + 1 = O(\log(m^\alpha))$, for a constant α. Throughout Sections 3 and 4, we identified a new and wide class of instances allowing for a polynomial time algorithm. We achieved this by showing how to build a dynamic programming algorithm which executes in time of $O(c^r \times m \times K)$ and takes space of $O(c^r \times m \times K)$ for the instances where $(c + 2u)/c$ is bounded by a constant r. These results represent a significant improvement to the understanding of the IKHO problem and more in general for the Interactive Knapsacks problems to which IKHO belongs and that are extensively presented by Aho in [4]. Note that our results imply Aho's results for IKHO, as shown in Section 2.2. In Section 5, we also exploited Hirschberg's approach in order to create a memory saving version of our algorithm which decreases the space complexity to $O(c^r \times K + m)$ without increasing the time complexity.

Extensive experimental evaluations have been performed on the c++ implementations of both algorithms, confirming the complexity estimates given in Sections 4 and 5. In addition, these experiments pointed out that, despite the heavy constants introduced in the time complexity bound of the memory saving version, the last version of the algorithm is often faster than the base version in the practice. This is mainly due to the fact that the higher memory usage of the base version brings the operative system to allocate data in the slowest memory devices like the RAM and the hard disk, instead of using the CPU cache.

Appendix

Proving the Base of Lemma 4

In this appendix, we prove the base of the induction of Lemma 4. We have to show that, assuming $c \geq e$, $S_c(n) \leq ((c+1)/c)c^{n/c}$ for $n \in [c, 2c]$.

In the following, let

$$F_c(i) = 1 + c + i + \frac{i(i-1)}{2}. \tag{A.1}$$

First, notice that $F_c(i) = S_c(c+i)$ for $i \in [0, c]$. Indeed, for $i \in [0, c]$, we can compute $S_c(c+i)$ by simply counting the number of signatures of length $c+i$, and then verify that this number is exactly $F_c(i)$. By Constraint (5), in a signature of length $2c$, or smaller than $2c$, there can be at most two insertions. Thus, since $c + i \leq 2c$, we can count the signatures of length $c+i$, by grouping them according to the number of insertions (no one, one, or two). Obviously, only the signature $\mathbf{0}^{c+i}$ has no insertions. Moreover, there are exactly $c+i$ ways to place one insertion. For the case where we have two insertions, note that the first one can be only in the first $i-1$ positions, because if we place it on the i-th position, by Constraint (5), there cannot be later insertions in a signature of length $i+c$. Moreover, notice that if we fix the first insertion in position j, then we have $i-j$ possible places where to put the second one, again by Constraint (5). Therefore, the number of signatures with two insertions is clearly

$$\sum_{j=1}^{i-1} i - j = \sum_{j=1}^{i-1} j = \frac{i(i-1)}{2}. \tag{A.2}$$

So $F_c(i) = S_c(c+i)$ for $i \in [0, c]$, as anticipated. Therefore, in order to prove the base of the induction of Lemma 4, it is enough to show that $F_c(i) \leq f_c(i)$, with $f_c(i) = ((c+1)/c)\ c^{(c+i)/c} = ((c+1)/c)c^{1+i/c} = (c+1)c^{i/c}$.

Clearly, for $i = 0$, we get $F_c(0) = c + 1 \leq (c+1)\ c^{0/c} = f_c(0)$. Our plan is then to observe that $\partial F_c(i)/\partial i \leq \partial f_c(i)/\partial i$ holds for every $i \geq 0$, when $c \geq e$. Indeed, $\partial F_c(i)/\partial i = i + 1/2$, whereas $\partial f_c(i)/\partial i = (c+1)\ c^{i/c}(\ln(c)/c)$. When $i = 0$, we get $F_c'(0) = 1/2$, and since $c \geq e$,

$$f_c'(0) = (c+1)\ c^0\ \frac{\ln(c)}{c} = \ln(c)\frac{c+1}{c} > 1. \tag{A.3}$$

That is, $F_c'(0) \leq f_c'(0)$. Moreover, it is simple to see that $F_c'(i)$ is linear, whereas $f_c'(i)$ is an exponential on the variable i, and, therefore, $f_c'(i) \geq F_c'(i)$ for every $i \geq 0$, as anticipated.

References

[1] Isto Aho, "Interactive Knapsacks," *Fundamenta Informaticae*, vol. 44, no. 1-2, pp. 1–23, 2000.

[2] Isto Aho, "On the approximability of interactive knapsacks problems," in *Proceedings of the 28th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM '01)*, vol. 2234 of *Lecture Notes in Computer Science*, pp. 152–159, Piešt'any, Slovak Republic, November/December 2001.

[3] Isto Aho, "New polynomial-time instances to various knapsack-type problems," *Fundamenta Informaticae*, vol. 53, no. 3-4, pp. 199–228, 2002.

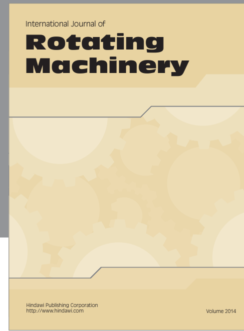
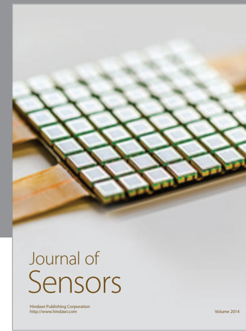
[4] Isto Aho, *Interactive Knapsacks: Theory and Application*, A-2002-13, University of Tampere, 2002.

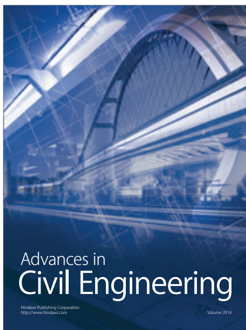
[5] E. Y.-H. Lin, "A bibliographical survey on some well-known non-standard knapsack problems," *INFOR*, vol. 36, no. 4, pp. 274–317, 1998.

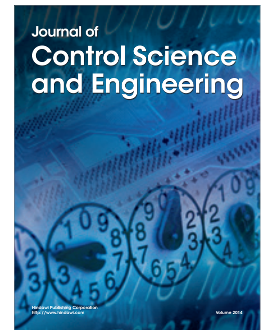
[6] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664–675, 1977.

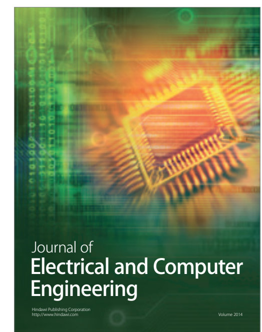
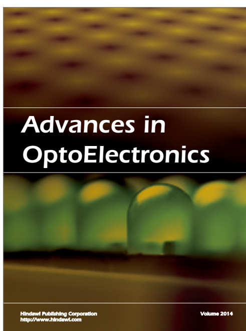
[7] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, Cambridge, UK, 1997.

Journal of
Engineering

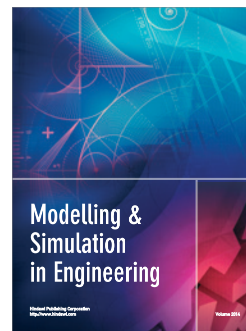
The Scientific
World Journal

International Journal of
Rotating
Machinery

Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

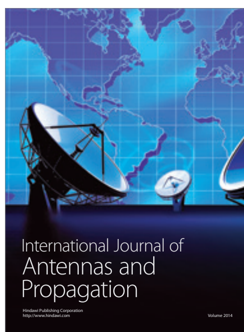
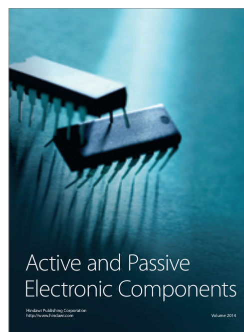
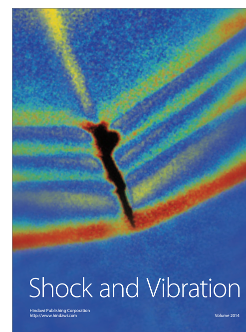
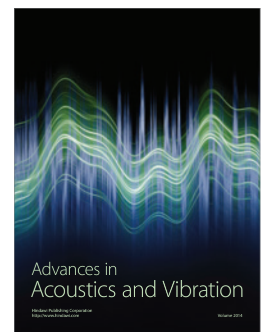
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration