Exact Approaches for the Knapsack Problem with Setups

Fabio Furini

Université Paris Dauphine, PSL Research University, LAMSADE, 75016 Paris, France
fabio.furini@dauphine.fr

Michele Monaci[1]

DEI, University of Bologna, 40136 Bologna, Italy
michele.monaci@unibo.it

Emiliano Traversi

Laboratoire d'Informatique de Paris Nord, Université de Paris 13, 93430 Villetaneuse, France,
emiliano.traversi@lipn.univ-paris13.fr

Abstract

We consider a generalization of the knapsack problem in which items are partitioned into classes, each characterized by a fixed cost and capacity. We study three alternative Integer Linear Programming formulations. For each formulation, we design an efficient algorithm to compute the linear programming relaxation (one of which is based on Column Generation techniques). We theoretically compare the strength of the relaxations and derive specific results for a relevant case arising in benchmark instances from the literature. Finally, we embed the algorithms above into a unified implicit enumeration scheme which is run in parallel with an improved Dynamic Programming algorithm to efficiently compute an optimal solution of the problem. An extensive computational analysis shows that our new exact algorithm is capable of efficiently solving all the instances of the literature and turns out to be the best algorithm for instances with a low number of classes.

Keywords: Knapsack Problems, Column Generation, Relaxations, Branch-and-Bound Algorithms, Computational Experiments.

1. Introduction

The classical *Knapsack Problem* (KP) is one of the most famous problems in combinatorial optimization. Given a knapsack capacity C and a set $N = \{1, \dots, n\}$ of items, the j-th having a profit p_j and a weight w_j, KP asks for a maximum profit subset of items whose total weight does not

[1]Corresponding author

exceed the capacity. This problem can be formulated using the follow Integer Linear Program (ILP):

$$\max \left\{ \sum_{j \in N} p_j x_j : \sum_{j \in N} w_j x_j \leq C, x_j \in \{0,1\}, j \in N \right\} \tag{1}$$

where each variable x_j takes value 1 if and only if item j is inserted in the knapsack.

KP is NP-hard, although in practice fairly large instances can be solved to optimality within low running time. The reader is referred to [16, 13] for comprehensive surveys on applications and variants of this problem.

In this paper we consider a generalization of KP arising when items are associated with operations that require some setup time to be performed. In particular, there is a given set $I = \{1, \ldots, m\}$ of *classes* associated with items, and each item j belongs to a given class $t_j \in I$. A non-negative *setup cost* f_i is incurred and a non-negative *setup capacity* s_i is consumed in case items of class i are selected in the solution. Without loss of generality, we assume that all input parameters have integer values. The resulting problem is known in the literature as Knapsack Problem with Setup (KPS).

KPS has been first introduced in the literature by [14] in a survey of non-standard knapsack problems worthy of investigation. In particular, this variant of KP was listed as it finds many practical application, e.g., when industries that produce several types of products must prepare some machinery related to the production of a certain class of products. In addition, it appears as a subproblem in scheduling capacitated machines, and may be used to model resource allocation problems. [10] designed a Lagrangean Decomposition for the setup knapsack problem, that may be seen as a variant of KPS in which the setup cost of each class and the profit associated to each item can take also negative values. The version of the problem in which only the setup cost for each class is taken into account, usually denoted as fixed charge knapsack problem, was addressed by [1] and [2]. In particular, the former presented an exact algorithm based on a branch-and-bound scheme, while the latter considered the case in which items can be fractionated by cross decomposition. The problem addressed by [18] is the multiple-class integer knapsack problem, a special case of KPS in which item weights are assumed to be a multiple of their class weight, and lower and upper bounds on the total weight of the used classes are imposed. For this problem, different ILP formulations were introduced and an effective branch-and-bound algorithm was designed. A Branch-and-Bound algorithm for KPS was given in [21]. This algorithm was tested on instances with up to 10.000 variables, and turned out to be effective mainly for instances where profits and weight are uncorrelated – while it ran out of memory for several large correlated instances. Motivated by an industrial application in a packing industry, KPS was studied by [5]; this article presented a basic dynamic programming scheme and an improved version of the algorithm, with a reduced storage requirement, that proved able to solve instances with up to 10000 items and 30 classes. Recently, KPS has also been addressed in [20] and [7]. The former introduces a new dynamic programming algorithm, gives negative results on the approximability of the problem in the general case, and considers some special cases for which fully polynomial time approximation schemes exist. The latter presents an exact approach for KPS based on the solution of several ILP models that show up to be easy to solve in practice. Computational experiments reported in [7], both on instances from the literature and on a large set of new randomly generated problems, show that, for many classes of problems, this approach is the state-of-the-art for the exact solution of KPS.

Paper Contributions. The contribution of the paper is twofold, as it embraces both theoretical and computational aspects. We develop linear-time algorithms for the optimal solution of the Linear Programming (LP) relaxation of different Integer Linear Programming formulations of KPS. Computational experiments show that these algorithms produce a considerable speedup with respect to the direct use of a commercial ILP solver. In addition, we derive for the first time an effective column generation approach to solve a KPS formulation with a pseudo-polynomial number of variables. Finally, we exploit these fast and strong relaxations within an unified branch-and-bound(-and-price) scheme. By reducing the space complexity of the Dynamic Programming algorithm proposed in [5], we managed to improve its computational performance. Since the new exact algorithms are particularly effective on complementary subsets of KPS instances, in order to deliver the best computational performance we proposed a parallel algorithm which exploit the qualities of all the new exact algorithms. We tested our new exact algorithms on a large set of instances proposed in the literature and on a new set of larger randomly generated problems. The outcome of our experiments is that the new approaches are competitive with the state-of-the-art exact algorithms for KPS, though they do not require the use of an ILP solver. In addition, we show that on some classes of instances, a considerable speedup may be obtained with respect to the other algorithms proposed so far in the literature.

In the rest of the paper we will denote by n_i the number of items in each class $i \in I$. We assume that $n_i \geq 2$ for some class $i \in M$ and $m > 1$; otherwise, one could associate the setup capacity and cost to the items, yielding a KP. For a similar reason, we can assume that $f_i > 0$ and/or $s_i > 0$ for some class $i \in I$. Without loss of generality, we assume that items are sorted according to their class, i.e., class i includes all items $j \in K_i := [\alpha_i, \beta_i]$, where $\alpha_i = \sum_{k=1}^{i-1} n_k + 1$ and $\beta_i = \alpha_i + n_i - 1$. Moreover, we assume for the presentation that, within each class, items are sorted according to non-increasing profit over weight ratio, i.e.,

$$\frac{p_j}{w_j} \geq \frac{p_{j+1}}{w_{j+1}} \quad j = \alpha_i, \ldots, \beta_i - 1; \quad i \in I.$$

To avoid pathological situations, we also assume that the cost of each class $i \in I$ is smaller than the total profit of its items, i.e., $f_i < \sum_{j \in K_i} p_j$, since otherwise this class will never be used in any optimal solution. We assume that not all items (and classes) can be selected, i.e., $\sum_{j \in J} w_j + \sum_{i \in I} s_i > C$; otherwise a trivial optimal solution is obtained by taking all items and classes. Finally, we assume that each item $j \in N$ satisfies $w_j + s_{t_j} \leq C$; otherwise item j cannot be inserted in any feasible solution, and can be removed from consideration.

Let us introduce a first numerical example, called *Example 1* in the following. This instance has 2 classes ($m = 2$) with two items each, i.e., $n = 4, n_1 = 2, \alpha_1 = 1, \beta_1 = 2, n_2 = 2, \alpha_2 = 3$ and $\beta_2 = 4$. The set up costs and capacities of the classes are $f_1 = 10, s_1 = 10, f_2 = 9$ and $s_2 = 6$. The profits and weights of the items are the following: $p_1 = 84, w_1 = 75, p_2 = 75, w_2 = 72, p_3 = 70, w_3 = 64, p_4 = 71$ and $w_4 = 78$. Finally, the knapsack capacity is $C = 152$. The optimal solution value of Example 1 is 132 and the corresponding solution takes both items of the second class. This example will be used to demonstrate some important properties of the KPS models in the following sections.

The paper is organized as follows: in Section 2 we introduce alternative formulations of KPS

and discuss the properties of the associated linear programming relaxations. In Section 3 we give efficient combinatorial algorithms for solving the LP relaxations of the models; these algorithms are embedded into an enumerative algorithm described in Section 4. In Section 5 we discuss some improvements to the dynamic programming algorithm proposed in [5]. Section 6 describes a relevant special case of KPS and shows the additional properties of the models in this case. Finally, Section 7 reports an extensive computational experience on the solution of the ILP models (and their relaxations) either using a general purpose solver or executing our algorithms and compares their performance with other approaches from the literature, and Section 8 draws some conclusions.

2. Integer Linear Programming models for KPS

In this section we introduce alternative formulations for KPS and discuss the relation between the associated linear programming relaxations. These formulations and the associated LP relaxations will be computationally tested in Section 7.

2.1. Model M1

A natural model for KPS is obtained by introducing x_j variables that have the same meaning as in (1), and decision variables y associated with item classes: in particular, each variable y_i takes value 1 iff some item of class i is included in the solution. The resulting model is as follows

$$\max \quad \sum_{j \in N} p_j \, x_j - \sum_{i \in I} f_i \, y_i \tag{2}$$

$$\sum_{j \in N} w_j \, x_j + \sum_{i \in I} s_i \, y_i \leq C \tag{3}$$

$$x_j \leq y_{t_j} \qquad j \in N \tag{4}$$

$$x_j \in \{0, 1\} \qquad j \in N \tag{5}$$

$$y_i \in \{0, 1\} \qquad i \in I. \tag{6}$$

The objective function (2) maximizes the total profit of the selected items minus the setup cost of the used classes, whereas constraint (3) takes into account that capacity is used both for the item weight and for the setup of the classes. Inequalities (4) force a class to be used whenever some item of the class is selected. Finally, (5)–(6) impose all variables be binary. It is worth mentioning that constraints (4)-(5) and the objective function force the y variables to be binary; thus, in principle, constraints (6) are redundant. The resulting model, denoted as M1 in the following, has $n + m$ variables and $n + 1$ constraints, plus variable domain constraints.

By replacing constraints (5)–(6) with the following ones:

$$x_j \in [0, 1] \qquad j \in N \tag{7}$$

$$y_i \in [0, 1] \qquad i \in I \tag{8}$$

we obtain the LP relaxation of M1, that will be denoted as LP1 in what follows. An effective combinatorial algorithm to solve LP1 is given in Section 3.

2.2. Model M2

In this section we present a lighter model, that contains fewer constraints than M1, obtained by replacing constraints (4) with the following ones

$$\sum_{j \in K_i} \overline{w}_j \, x_j \quad \leq C_i^{\overline{w}} \, y_i \quad i \in I. \tag{9}$$

Constraints (9) link x and y variables and represent a surrogate relaxation of constraints (4), using non-negative *surrogate weights* $\overline{w}_j$ ($j \in N$). For each class $i \in I$ coefficient $C_i^{\overline{w}}$ can be defined as follow

$$C_i^{\overline{w}} = \max \left\{ \sum_{j \in K_i} \overline{w}_j \theta_j : \sum_{j \in K_i} w_j \theta_j \leq C - s_i, \theta_j \in \{0,1\}, \; \forall j \in K_i \right\}, \tag{10}$$

i.e., it can be computed solving a KP with profits and weights defined by the surrogate and original weights, respectively. The capacity of this KP can be set to $C - s_i$ in order to take into account the setup capacity, if some item of class i is selected.

The mathematical model defined by (2)-(3)-(5)-(6) and (9) will be denoted as M2, and corresponds to a family of valid formulations for KPS, defined according to weights $\overline{w}$.

The first natural choice for $\overline{w}$ is to use the original item weights, obtaining the following surrogate constraints:

$$\sum_{j \in K_i} w_j \, x_j \quad \leq C_i^{\overline{w}} \, y_i \quad i \in I. \tag{11}$$

A second alternative is to use unitary surrogate weights $\overline{w}$:

$$\sum_{j \in K_i} x_j \quad \leq C_i^{\overline{w}} \, y_i \quad i \in I. \tag{12}$$

Summarizing, we considered 2 different variants of model M2, obtained by using different values of $\overline{w}_j$ for each item $j \in N$, namely:

- M2_A: $\overline{w}_j = w_j$, i.e., reducing constraints (9) to (11);
- M2_B: $\overline{w}_j = 1$, i.e., reducing constraints (9) to (12).

The formulation above has the same number of variables as M1, but only $m + 1$ constraints (instead of $n + 1$). We will denote by LP2 the linear programming relaxation of model M2, i.e., the problem defined by (2), (3), (9), (7) and (8). The following result shows that there is no dominance between LP1 and LP2.

Observation 1. *There is no dominance between models M1 and M2 in terms of linear programming relaxation.*

Proof. We show the thesis by giving two numerical instances for which the bounds exhibit opposite behavior. Consider the instance of Example 1. An optimal solution for LP1 is given by $x_1^* = x_2^* = y_1^* = 0.968153$ and has value 144.254. Model M2_A has $C_1^{\overline{w}} = 75$ and $C_2^{\overline{w}} = 142$. An optimal solution of its LP relaxation is $x_1^* = x_3^* = 1, x_4^* = 0.003638$ and $y_1^* = 1, y_2^* = 0.452703$, yielding an upper bound equal to 140.183. Model M2_B has $C_1^{\overline{w}} = 1$ and $C_2^{\overline{w}} = 2$, and an optimal solution of its LP relaxation is $x_1^* = x_3^* = 1$ and $y_1^* = 1, y_2^* = 0.5$, with value 139.5.

Conversely, consider an instance for which $C_i^{\overline{w}} > \overline{w}_j \geq 1$ for some item $j \in K_i$. The solution, say (x^*, y^*), with $x_j^* = 1/\overline{w}_j$, $y_i^* = 1/C_i^{\overline{w}}$, and all other x and y variables set to 0 is feasible for LP2. However, $C_i^{\overline{w}} > \overline{w}_j$ implies $y_i^* < x_j^*$, i.e., the solution is not feasible for LP1 since it violates the associated constraint (4). □

Observe that the result above is valid in case coefficients $C_i^{\overline{w}}$ in M2 are computed according to (10), which requires the solution of a KP.

2.3. Model M3

In this section we present an extended model which contains an exponential number of variables. Let us introduce the following collections $\mathscr{S}_i$ ($i \in I$) of feasible subsets of items $S \subseteq K_i$ satisfying the knapsack capacity C

$$\mathscr{S}_i = \left\{ S \subseteq K_i : \sum_{j \in S} w_j \leq C - s_i \right\}.$$

For each item subset $S \in \mathscr{S}_i$, we can define its profit and weight taking also into account the setup cost and capacity of the corresponding class $i(S)$:

$$P_S = \sum_{j \in S} p_j - f_{i(S)}, \quad W_S = \sum_{j \in S} w_j + s_{i(S)}.$$

A valid model for KPS can be obtained by introducing, for each subset $S \in \mathscr{S}_i$ ($i \in I$), a binary variable ξ_S which takes value 1 iff subset S is included in the solution:

$$\max \quad \sum_{i \in I} \sum_{S \in \mathscr{S}_i} P_S \xi_S \tag{13}$$

$$\sum_{i \in I} \sum_{S \in \mathscr{S}_i} W_S \xi_S \leq C \tag{14}$$

$$\sum_{S \in \mathscr{S}_i} \xi_S \leq 1 \qquad i \in I \tag{15}$$

$$\xi_S \in \{0, 1\} \qquad i \in I, S \in \mathscr{S}_i. \tag{16}$$

Objective function (13) maximizes the total profit of the selected subset of items, whereas constraint (14) ensure that the solution satisfies the capacity constraint. Inequalities (15) impose that at most

one subset is selected for each class, whereas constraints (16) impose all variables be binary. The resulting formulation, denoted as M3 in the following, corresponds to the classical formulation of the Multiple-Choice Knapsack Problem (MCKP) with inequality constraints; see [12].

By replacing constraints (16) with the following ones:

$$\xi_S \geq 0 \qquad i \in I, S \in \mathscr{S}_i, \tag{17}$$

we obtain the linear programming relaxation of M3, that will be denoted as LP3 in what follows. Note that constraints (15) implicitly provide an upper bound of value 1 on the ξ_{S_i} variables, thus we do not need to impose this bound in (17).

Finally, we observe that the model above has already been proposed by [4] and used, e.g., by [18] to derive an exact approach. In addition, in both papers above, the authors observed that the set of variables in the model can be reduced to a pseudo-polynomial number, considering at most one variable for each item class and possible value of capacity. Since this would require a huge number of variables in large instances, in our approach we used the model above by generating variables on-the-fly, according to a column generation scheme (see Section 3.3).

The quality of the upper bound obtained solving the LP relaxation of M3 cannot be worse than its counterpart associated with models M1 and M2:

Observation 2. *Model M3 dominates both models M1 and M2 in terms of linear programming relaxation.*

Proof. We first show that any feasible solution for LP3 can be converted in a solution that is feasible for both LP1 and LP2. Let ξ^* denote a feasible solution to LP3 and define a solution (x^*, y^*) as follows: for each class i set

$$y_i^* = \sum_{S \in \mathscr{S}_i} \xi_S^* \quad \text{and} \quad x_j^* = \sum_{\substack{S \in \mathscr{S}_i \\ j \in S}} \xi_S^* \quad (j \in K_i).$$

The total weight used by class i in this solution is equal to

$$s_i y_i^* + \sum_{j \in K_i} w_j x_j^* = s_i \sum_{S \in \mathscr{S}_i} \xi_S^* + \sum_{j \in K_i} w_j \sum_{\substack{S \in \mathscr{S}_i \\ j \in S}} \xi_S^* = \sum_{S \in \mathscr{S}_i} (s_i + \sum_{j \in S} w_j) \xi_S^* = \sum_{S \in \mathscr{S}_i} W_S \xi_S^*.$$

Thus, inequality (14) ensures that the capacity constraint is satisfied. Observe that, by construction, each item $j \in K_i$ has $x_j^* \leq y_i^*$; thus, (x^*, y^*) is feasible to LP1. To show that (x^*, y^*) for feasible to LP2 as well, it is enough to note that for each class i we have

$$\sum_{j \in K_i} \overline{w}_j x_j^* = \sum_{j \in K_i} \overline{w}_j \sum_{\substack{S \in \mathscr{S}_i \\ j \in S}} \xi_S^* = \sum_{S \in \mathscr{S}_i} \xi_S^* \sum_{j \in S} \overline{w}_j \leq \sum_{S \in \mathscr{S}_i} \xi_S^* C_i^{\overline{w}} = C_i^{\overline{w}} \sum_{S \in \mathscr{S}_i} \xi_S^* = C_i^{\overline{w}} y_i^*$$

where the inequality is valid for each feasible item set $S_i \in \mathscr{S}_i$ due to the definition of $C_i^{\overline{w}}$, see (10).

Consider now the instance of Example 1. The optimal solution of LP3 is $\xi_{S_1}^* = 0.047$ $\xi_{S_2}^* = 1$ where the two subsets are $S_1 = \{2\}$ and $S_2 = \{3, 4\}$, and belong to classes 1 and 2, respectively. For these item sets we have $P_{S_1} = 74, P_{S_2} = 132, W_{S_1} = 85$ and $W_{S_2} = 148$. Thus, the optimal solution value is 135.482, i.e., it is lower than the value of the LP relaxations of M1, M2_A and M2_B which are 144.254, 140.183 and 139.5, respectively (see Observation 1). □

3. Efficient computation of upper bounds for KPS

A natural way to compute an upper bound on the optimal solution value of a KPS instance is to solve the LP relaxation of the models introduced in Section 2 using some general LP solver. In this section we present effective combinatorial algorithms for solving the LP relaxation of the models above with no need of an external LP solver.

3.1. Solving the LP relaxation of M1

In this section we consider the LP relaxation of model M1. We observe that [21] introduces some properties of an optimal solution to LP1 and derives a combinatorial algorithm for its solution. However, no analysis on the computational complexity of the algorithm is given in [21]. Thus, for the sake of completeness, we first report the relevant properties of an optimal solution of LP1, and then present the solution algorithm and analyze its time complexity. This algorithm extends the one given by [1] for the special case in which only the setup costs are addressed, and improves the time complexity with respect to that claimed in [1]. Finally, we propose a strengthened relaxation that can be computed in constant time.

To simplify the notation, we introduce, for each item $j \in K_i$, the following quantities

$$P(j) = \sum_{h=\alpha_i}^{j} p_h - f_i \quad \text{and} \quad W(j) = \sum_{h=\alpha_i}^{j} w_h + s_i$$

These figures refer to cumulative profit and weight, respectively, that would be obtained taking the all items of class i up to item j, and take into account setup cost and capacity of the class as well.

Definition 1. *Consider a given class $i \in I$, and let*

$$\overline{K}_i = \{j : \frac{P(j)}{W(j)} > \frac{p_{j+1}}{w_{j+1}} : j = \alpha_i, \ldots, \beta_i - 1\}.$$

We define the break item *of class i*

$$b_i = \begin{cases} \min\{j \in \overline{K}_i\} & \textit{if } \overline{K}_i \neq \emptyset \\ \beta_i & \textit{otherwise} \end{cases} \tag{18}$$

Intuitively, the break item of class i is the first item in K_i (if any) for which the ratio between the cumulative profit and the cumulative weight is larger than the profit over weight ratio of all subsequent items, i.e.,

$$\frac{P(j)}{W(j)} \leq \frac{p_{j+1}}{w_{j+1}} \;\; \forall j = \alpha_i, \ldots, b_i - 1 \qquad \text{and} \qquad \frac{P(b_i)}{W(b_i)} > \frac{p_{b_i+1}}{w_{b_i+1}} \tag{19}$$

If no such item exists, the break item is conventionally defined as the last item of the class—and the second inequality (19) is not defined.

Theorem 1. *For each class $i \in I$, the break item b_i can be computed in $O(n_i)$ time.*

Proof. The algorithm that determines the break item for a given class is similar to the scheme proposed by [3] for finding the critical item in linear time in a KP instance. The algorithm performs a number of iterations computing, at each step, the median of m elements in $O(m)$ time, and removing at least half of the elements from consideration for the next iteration. This yields an overall time complexity of the algorithm equal to $O(n_i)$. The detailed algorithm is given in the Appendix. □

The following results is implicit in [21]. For the sake of completeness we report it and give a proof in the Appendix.

Theorem 2. *[21] There exists an optimal solution x^*, y^* of LP1 that fulfills the following properties for each item class $i \in I$*

1. $y_i^* = \max\{x_j^* : j \in K_i\}$;
2. $x_j^* = y_i^* \quad \forall j = \alpha_i, \ldots, b_i$.

Theorem 2 states that an optimal solution to LP1 exists in which, for each class i, all variables associated with items in set $\{\alpha_i, \ldots, b_i\}$ as well as variable y_i take the same value. Thus, we one can replace all such items with a single *cumulative item* I_i with profit $\overline{P}_i$ and weight $\overline{W}_i$, where

$$\overline{P}_i = P(b_i) = \sum_{h=\alpha_i}^{b_i} p_h - f_i \qquad \text{and} \qquad \overline{W}_i = W(b_i) = \sum_{h=\alpha_i}^{b_i} w_j + s_i. \tag{20}$$

Each cumulative item takes into account the setup capacity and cost of its class, and has a profit over weight ratio better than the remaining items of the class, if any. Thus, an optimal solution to LP1 can be obtained applying the well-known Dantzig's algorithm (see [6]) to the KP instance defined by cumulative items and all items $j = b_i + 1, \ldots, \beta_i$ for each class i. The resulting algorithm, described in Figure 1, considers one (either original or cumulative) item at a time, and inserts the current item if it fits in the residual capacity; otherwise, the *critical item* is found, and only a fraction of the item is inserted in the knapsack. Note that, for each class i, all items after the break item have a profit over weight ratio that is worse than that of the cumulative item; thus, they may be inserted in the knapsack only after the cumulative item is packed (i.e., after the setup cost and capacities are incurred). Observe also that, though many items may be taken at a fractional value, at most one y variable may be fractional, similarly to what happens in the solution of the LP relaxation of KP.

Theorem 3. *An optimal solution to LP1 can be computed in $O(n)$ time.*

Proof. As proved in Theorem 1 the set of break items can be computed in overall $O(n)$ time, which allows the definition of the knapsack instance in linear time. This instance includes at most n items. Hence, its LP relaxation can be computed in $O(n)$ time, using again the procedure by [3] for finding the critical item and applying Dantzig's algorithm. □

We conclude this section showing a strengthened relaxation that exploits the fact that the optimal LP1 solution has at most one fractional item and that, in any integer solution, this variable must take either value 0 or 1. Let $p(t)$ and $w(t)$ denote the profit and the weight, respectively, of the (either

Algorithm *LP1*:

initialize: $\overline{N} := \emptyset$;
for each class $i \in I$ **do**
    compute the break item b_i;
    define the cumulative item $\overline{I}_i$, according to (20);
    $\overline{N} := \overline{N} \cup \{I_i\} \cup \{b_i + 1, \ldots, \beta_i\}$;
end do
Solve the LP relaxation of the KP instance defined by item set $\overline{N}$, and let θ be the associated solution;
for each class $i \in I$ **do**
    set $y_i^* = \theta_{I_i}$ and $x_j^* = \theta_{I_i} \quad \forall j = \alpha_i, \ldots, b_i$;
    set $x_j^* = \theta_j \quad \forall j = b_i + 1, \ldots, \beta_i$;
end do

Figure 1: Algorithm to compute an optimal solution to LP1.

original or cumulative) item that is selected at each iteration t; in addition, let $\bar{t}$ be the number of iterations executed by the algorithm and $\bar{c}$ denote the residual capacity before inserting the last item. Similar to the MT bound proposed for KP by [15], we can derive the following upper bound for KPS

$$\overline{UB} = \max\{UB_0, UB_1\} \tag{21}$$

where

$$UB_0 = \sum_{t=1}^{\bar{t}-1} p(t) + \bar{c}\frac{p(\bar{t}+1)}{w(\bar{t}+1)} \quad \text{and} \quad UB_1 = \sum_{t=1}^{\bar{t}-1} p(t) + \left(p(\bar{t}) - (w(\bar{t}) - \bar{c})\frac{p(\bar{t}-1)}{w(\bar{t}-1)}\right)$$

represent an upper bound on the optimal solution value when the fractional item is fixed to 0 and 1, respectively. In case $\bar{t} = 1$, this improved bound cannot be computed. Otherwise, it can be easily seen that $\overline{UB}$ dominates the bound produced by LP1 and that the computational effort for computing this bound is negligible if an optimal solution to LP1 has been computed.

3.2. Solving the LP relaxation of M2

Similar to Section 3.1, we will compute an upper bound on the optimal solution of a KPS instance with a combinatorial algorithm based on the LP relaxation of model M2. In particular, we will denote by RLP2 the relaxation by LP2 removing the upper bound on variables y_i, i.e., replacing constraints (8) with $y_i \geq 0 \quad (i \in I)$.

Observation 3. *There exists an optimal solution of RLP2, say* (x^*, y^*)*, such that*

$$y_i^* = \frac{\sum_{j \in K_i} \overline{w}_j x_j^*}{\overline{C}_i^{\overline{w}}} \quad \forall i \in I \tag{22}$$

Proof. For a given class i, constraint (9) imposes the above lower bound for variable y_i^*. It is clear that increasing y_i^* with respect to this value produces a decrease of the solution value, unless $f_i = s_i = 0$. □

Based on Observation 3 one can reformulate RLP2 by substituting y variables; this yields to the following model

$$\max\left\{\sum_{j\in N}\widetilde{p}_j x_j : \sum_{j\in N}\widetilde{w}_j x_j \le C, x_j \in [0,1], j \in N\right\},$$

where

$$\widetilde{p}_j = p_j - \frac{f_{t_j}}{\overline{C}^{\overline{w}}_{t_j}}\overline{w}_j \quad \text{and} \quad \widetilde{w}_j = w_j + \frac{s_{t_j}}{\overline{C}^{\overline{w}}_{t_j}}\overline{w}_j. \tag{23}$$

This model corresponds to the LP relaxation of a knapsack problem, which can efficiently be solved using again Dantzig's algorithm in linear time. Once an optimal solution, say x^*, is computed for the relaxation above, variables y^* can be computed a posteriori according to (22).

We conclude this section observing that, to the best of our knowledge, no combinatorial algorithms are available to solve LP2, whose solution requires instead the use of an LP solver. However, the quality of the bound associated to LP2 is comparable with that obtained solving RLP2, for which our combinatorial algorithm is available. Finally, we mention that another relaxation of model M2 exists in which the integrality constraint is dropped for y variables only, while x variables are required to be binary; by definition this relaxation dominates RLP2. Solving this relaxation requires the solution of a KP (NP-hard problem), possibly defined by non-integer profits and weights. Extensive computational tests show that this relaxation produces only marginal improvements, while the computational effort for solving the relaxation can be considerably larger than for RLP2.

3.3. Solving the LP relaxation of M3

Model M3 has exponentially many ξ_S variables ($i \in I, S \in \mathscr{S}_i$), which cannot be explicitly enumerated for large-size instances. *Column Generation* (CG) techniques are then necessary to efficiently solve its linear programming relaxation. In the following we discuss the CG framework for M3 only, and refer the interested reader to [8] for further details on CG.

Model (13)–(15) and (17), initialized with a subset of variables containing a feasible solution, is called *Restricted Master Problem* (RMP). Additional new variables, needed to solve LP3 to optimality, can be obtained by separating the following dual constraints:

$$W_S\lambda + \pi_i \ge P_S \qquad\qquad i \in I, S \in \mathscr{S}_i, \tag{24}$$

where π_i ($i \in I$) is the dual variable associated with the i-th constraint (15) and λ is the dual variable associated with constraint (14). Accordingly, CG performs a number of iterations, until no violated dual constraint exist. At each iteration, the so-called *Pricing Problem* (PP) associated with each class $i \in I$ is solved. This problem asks to determine (if any) a subset $S^* \in \mathscr{S}_i$ for which the associated dual constraint (24), is violated, i.e., such that

$$\sum_{j\in S^*}(p_j - \lambda^* w_j) > \pi_i^* + \lambda^* s_i + f_i, \tag{25}$$

where π_i^* ($i \in I$) and λ^* are the dual variables values associated to the current solution of the RMP.

The pricing problem for class i asks for determining a subset of items $S^* \in \mathscr{S}_i$ that maximizes the left-hand-side of (25), and checking if this is larger than $\pi_i^* + \lambda^* s_i + f_i$. As such, finding the maximally violated dual constraint can be modeled as a KP, where each item $j \in K_i$ has profit $p_j - \lambda^* w_j$ and weight w_j. Using binary variables θ_j ($j \in K_i$), the problem reads as follows:

$$\tau^* = \max \left\{ \sum_{j \in K_i} (p_j - \lambda^* w_j)\theta_j : \sum_{j \in K_i} w_j \theta_j \leq C - s_i, \theta_j \in \{0,1\},\ \forall j \in K_i \right\}, \tag{26}$$

where $\theta_j = 1$ iff item j belongs to subset S^*. All variables with negative reduced costs that are generated, i.e., such that $\tau^* > \pi_i^* + \lambda^* s_i + f_i$ (if any), are added to the RMP, which is then re-optimized, according to a classic column generation scheme. If no column with negative reduced cost exists, the RMP is optimally solved and its solution (value) corresponds to the linear programming relaxation (value) of M3.

We already showed that the pricing problem asks for the solution of a KP for each class, which makes the solution of LP3 weakly NP-hard. We now describe a combinatorial algorithm that may be used at each iteration to compute both an optimal solution to RMP and an associated dual solution (which is required for solving the pricing problems). Since classical algorithms from the literature for MCKP address the problem in which constraints (15) are imposed as equalities (see, e.g., [13]), we introduce, for each class $i \in I$, a dummy subset S_d^i with $P_{S_d^i} = 0$ and $W_{S_d^i} = 0$.

From a given instance of M3, a corresponding instance of MCKP can be defined by using the same set of classes I and by introducing for each subset $S \in \mathscr{S}_i$ ($i \in I$) an item of weight W_S and profit P_S. To avoid misunderstanding in the following, we use the term subsets to refer to items of the MCKP and to distinguish them from items of the original KPS problem. Hence, we identify with ξ_S the variable associated to a given subset S.

The algorithm described in [13] operates in two steps: a first preprocessing phase, where dominated subsets are excluded due to consideration on their weight and profit, and a second phase, in which the residual subsets are sorted and added to the solution up to the completion of the total capacity. The preprocessing eliminates some subsets with pairwise and triplet-wise comparison. The discarted subsets are (I)LP-dominated, i.e., they will never appear in an optimal (integer) linear programming solution. For more detail about the elimination of dominated subsets, we refer the reader to [13].

Then the algorithm sorts for each class the subsets according to increasing value of their weights. For a given subset S, we indicate with $S-1$ the subset (belonging to the same class) immediately preceding S in the ordering (if any). After the sorting, to each subset S (starting from the second one) is associated a *slope(S)* value:

$$\frac{P_S - P_{S-1}}{W_S - W_{S-1}}.$$

which measures the ratio between the incremental profit gained by substituting subset S with subset $S-1$ in the solution and the associated incremental weight that is required. The elimination of the dominated subsets also implies that, for a given triplet of subsets S', S'' and S''', with

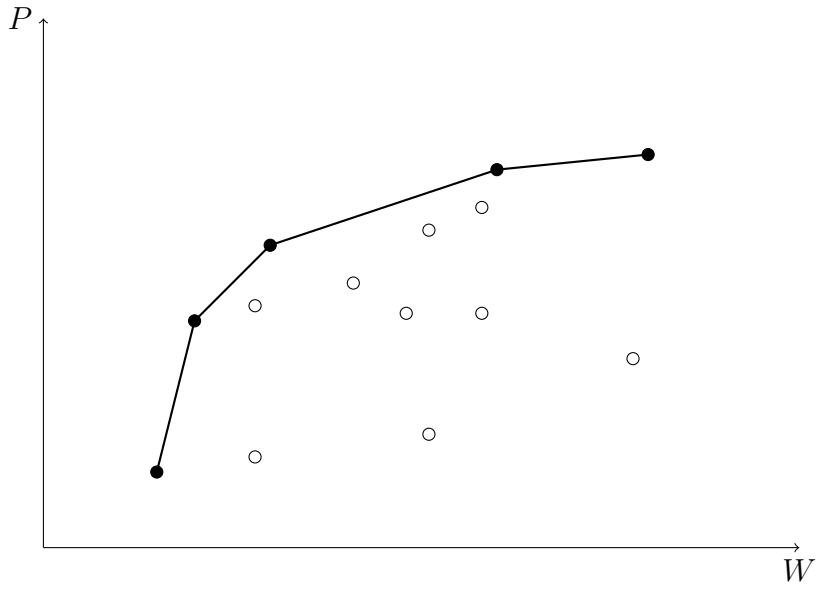

Figure 2: Example of undominated subsets (black dots) and dominated subsets (white dots).

$W_{S'''} \geq W_{S''} \geq W_{S'}$ we have

$$\frac{P_{S'''} - P_{S'}}{W_{S'''} - W_{S'}} \geq \frac{P_{S'''} - P_{S''}}{W_{S'''} - W_{S''}} \ .$$

Figure 2 shows an example of how the subsets of a given class look like after the preprocessing and the reordering. In the figure, each dot is associated a subset, plotted according to its weight and profit: white dots represent the dominated (and hence eliminated) subsets, while black dots represent the remaining, undominated, subsets. It is important to notice that, as byproduct of the elimination of the dominated subsets, the remaining subsets are also ordered according to a decreasing value of their slope.

The algorithm starts with a feasible MCKP solution containing the first subset of each class (i.e. the ones of minimum weight) and computes the associated residual capacity C_r and solution profit z. At each iteration, the algorithm improves the solution by (i) determining the subset with higher slope, say S', among all classes; (ii) replacing subset $S' - 1$ with subset S'; and (iii) updating the residual capacity $C_r = C_r - W_{S'} + W_{S'-1}$ and solution profit $z = z + P_{S'} - P_{S'-1}$. The algorithm stops when a subset $\hat{S}$ that does not fit in the knapsack is found, i.e., such that $W_{\hat{S}} - W_{\hat{S}-1} > C_r$. In this case, subsets $\hat{S}$ and $\hat{S} - 1$ are packed in the optimal solution with a fractional value, as follows

$$\xi_{\hat{S}} = \frac{C_r}{W_{\hat{S}} - W_{\hat{S}-1}} \qquad\qquad \xi_{\hat{S}-1} = 1 - \xi_{\hat{S}} \tag{27}$$

and the algorithm terminates. In the following we will refer to $\hat{S}$ as the *critical* subset. In case no critical subset exists, we will use the term to denote the first subset that is excluded in the solution. Given this definition, we characterize an optimal dual solution to MCKP as follows:

Theorem 4. *Let $\hat{S}$ be the critical subset in an optimal primal solution of LP3. Then, an optimal solution to the associated dual is the following:*

$$\lambda^* = \frac{P_{\hat{S}} - P_{\hat{S}-1}}{W_{\hat{S}} - W_{\hat{S}-1}} \qquad\qquad \pi_i^* = \max_{S \in \mathscr{S}_i} \{P_S - W_S \lambda^*\} \ .$$

Proof. It is easy to verify that (λ^*, π^*) satisfies the dual constraints. We hence need to show that the primal and dual solutions have same objective value.

Let us denote by $\hat{i}$ the class associated with the critical subset $\hat{S}$. The value of the optimal primal solution can be written as follows:

$$\sum_{i \in I} \sum_{S \in \mathscr{S}_i} P_S \xi_S = \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} P_S \xi_S + P_{\hat{S}-1} \xi_{\hat{S}-1} + P_{\hat{S}} \xi_{\hat{S}} = \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} P_S \xi_S + P_{\hat{S}-1} + (P_{\hat{S}} - P_{\hat{S}-1}) \xi_{\hat{S}}$$

$$= \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} P_S \xi_S + P_{\hat{S}-1} + \frac{C_r (P_{\hat{S}} - P_{\hat{S}-1})}{W_{\hat{S}} - W_{\hat{S}-1}} = \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} P_S \xi_S + P_{\hat{S}-1} + C_r \lambda^*$$

Since $C_r = C - \left(\sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} W_S \xi_S + W_{\hat{S}-1} \right)$ we have

$$\sum_{i \in I} \sum_{S \in \mathscr{S}_i} P_S \xi_S = \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathscr{S}_i} (P_S - \lambda^* W_S) \xi_S + C\lambda^* + (P_{\hat{S}-1} - \lambda^* W_{\hat{S}-1}) \leq \sum_{i \in I} \pi_i^* + C\lambda^*$$

where the latter inequality derives from the definition of π_i^* variables and from the fact that, for each class i, exactly one subset is selected in the primal solution. As the objective function of the dual is $C\lambda + \sum_{i \in I} \pi_i$, the weak duality theorem ensures that (π_i^*, λ^*) is an optimal dual solution. □

4. Exact solution of KPS

In this section we describe an exact approach to KPS based on branch-and-bound techniques. Section 4.1 describes the way each branching node is evaluated, whereas Section 4.2 shows how a local upper bound is computed at each branching node.

4.1. Node exploration

Our enumerative algorithm is based on the observation that KPS reduces to KP in case the set of item classes to be selected is given. This suggests a branching rule in which first-stage decisions are associated with the classes, whereas variables associated with items are treated as second-stage variables. For the sake of simplicity, in this section we will refer to the first formulation of KPS, i.e., we will make use of x and y variables to refer to selection of items and classes, respectively.

Figure 3 reports the pseudocode of the algorithm that is executed at each node of the tree. We first solve the LP relaxation at the current node and check whether the node can be fathomed, comparing the local upper bound with the incumbent solution, say z^*. In case enumeration must continue,

we use a branching scheme similar to the one proposed by [11] for KP: at the root node we sort the classes according to non-increasing profit over weight ratio of the associated cumulative item, see (20). At each node, we take the first y variable that is not fixed by branching and define two descendant nodes by fixing this variable to 1 and 0, respectively. Subsequent nodes, if any, are explored in the order they are generated, according to a depth-first strategy. Finally, if all the y variables are fixed by branching, a backtracking is executed.

Note that in case all the y^* variables are integer, a heuristic step is executed to determine the optimal solution of the KP instance defined by all items in the selected classes. This is not strictly required for the correctness of the algorithm: indeed, an alternative strategy exists in which the resulting KP instance is solved only at the leaf nodes of the branch-and-bound tree. However, our computational experience showed a degradation in the performances of the resulting algorithm, which is able to update the incumbent solution very rarely. Though KP is an NP-hard problem, effective codes for its solution can be found in the literature; in our implementation we used the routine `combo` proposed by [17], which is the state-of-the-art for KP problems with integer data. Obviously this step, that allows to avoid explicit branching on the x variables, is not required if all the x^* variables are integer as well; in this case, the incumbent is updated and a backtracking is performed.

Finally observe that our scheme may require to branch on y variables also in case all of them take an integer value in the current LP solution. The following example shows that this (apparently,

Algorithm *Solve_node*:

// LP solution and possible fathoming
solve the LP relaxation at the current node;
let (x^*, y^*) denote an optimal LP solution, and U the associated value;
if $U \leq z^*$ **then** fathom the node and return;
else
    // possible heuristic solution
    if all y^* variables are integer **then**
        solve a KP instance defined by items in the selected classes;
        let $z(KP)$ be the associated profit (including setup costs);
        if $z(KP) > z^*$ **then**
            update $z^* := z(KP)$;
            if $U \leq z^*$ **then** fathom the node and return;
        endif
    endif
    // possible branching
    if all y variables are fixed by branching **then** fathom the node and return;
    else
        let i be the first class that is not fixed by branching;
        define two subproblems branching on variable y_i;
    endif
return

Figure 3: Exploration of a branch-and-bound node.

unnatural) branching is necessary to ensure the correctness of the approach. For a given $M \geq 4$, consider an instance defined by two item classes, both having a unitary setup cost and capacity. The first class includes two items with $p_1 = M$, $w_1 = 1$, $p_2 = M$, $w_2 = M$, whereas the second class includes one item only, with $p_3 = 2$ and $w_2 = 2$. The knapsack capacity is equal to 5. The LP relaxation of M1 has $y_1 = 1$, $x_1 = 1$ and $x_2 = 3/M$, while the second class is not used, i.e., $y_2 = x_3 = 0$. The associated upper bound is $(M - 1) + 3 = M + 2$. On the contrary the optimal integer solution is $y_1 = y_2 = 1$, wit items $x_1 = x_3 = 1$ and $x_2 = 0$; the optimal integer value is M.

As already observed by [21], the algorithm by [1] does not allow branching on integer variables, and may thus fail in finding the optimal solutions in situations similar to the one depicted above.

4.2. Local upper bounds

In this section we describe the way in which the LP relaxation of the models above are solved at each node of the enumeration tree.

Solution of LP1. As branching conditions involve y variables only, the algorithm described in Section 3.1 for solving LP1 has to be modified as follows. At the root node we store all the original and cumulative items, sorted according to profit over weight ratio. At the current node, the local upper bound can be computed simply scanning the list of $n + m$ items: cumulative items can be used only for classes that are not fixed by branching. Original items can be used only for items that have been selected by branching (i.e., such that $y_i = 1$), while items that belong to a class that is forbidden by branching should not be used in the solution. It is easy to check that the computation of the local LP solution takes $O(n)$ time as at the root node.

Solution of LP2. Similar to LP1, solving RLP2 to optimality at each node requires small modification: items of classes that have been fixed to zero must not be selected, whereas for classes that have been selected, the fixed cost and capacity have to be taken into account, and items have to evaluated according to original profits and weights. Finally, for items that belong to the remaining classes one has to use profit and weights $\widetilde{p}_j$ and $\widetilde{w}_j$, respectively, see (23). Observe that this bound can still be computed in $O(n)$ time at each node after the root. In particular, one can define a copy of each item j with profit $\widetilde{p}_j$ and weight $\widetilde{w}_j$, to be used for evaluating item j in case the associated class t_j has not been fixed by branching. This doubled set of items is sorted at the beginning of the algorithm. At each node of the tree, one can scan this double list and insert only copies of items in classes that have not been fixed and items in classes that have been selected.

Solution of LP3. The same branching scheme can be used with LP3 as well. Since the y variables are not explicitly considered, the branching decision for a specific class i can be imposed changing the right-hand-side of the associated constraint (15) in M3. To impose the condition $y_i = 1$, the constraint becomes:

$$\sum_{S \in \mathscr{S}_i} \xi_S = 1. \tag{28}$$

On the other side, to impose the condition $y_i = 0$, the constraint becomes:

$$\sum_{S \in \mathscr{S}_i} \xi_S = 0. \tag{29}$$

These modifications do not change the nature of the formulation nor the associated pricing problems PP. The effect of constraint (28) is just to remove the non-negativity constraint on the corresponding dual variable. From a practical viewpoint, imposing $y_i = 0$ corresponds to disregard item class i and all the associated items; this makes LP3 easier to solve, since a smaller number of pricing problems has to be solved at each iteration of the column generation process (see Section 3.3). Finally, since new variables may be generated within the branching nodes, the branch-and-bound algorithm becomes in this case a *branch-and-price* algorithm.

5. An Improved Dynamic Programming Algorithm

In this section we describe a way of improving the storage requirements and the computational performances of the Dynamic Programming algorithms proposed in [5]. The basic algorithm given in [5] consists of n stages, each having 2 states associated with each possible capacity value from 0 to C. Let j be an item of class i, i.e., $j \in K_i$. For each capacity value r, state $A(j, r)$ reports the optimal solution value of the sub-instance defined by item set $\{1, \ldots, j\}$ and capacity r when class i is used, while state $B(j, r)$ gives the same figure when class i is not used (and the associated items cannot be selected). This scheme requires to store two matrices of size $n \times (C + 1)$, which makes the algorithm not suitable for instances with a large value of capacity and/or large numbers of items. To reduce the storage requirements, a second scheme was proposed in [5], that converts a KPS solution into an integer index. However, this can produce some slow-down in the performances of the approach. We refer the interest reader to [5] for further details on these two algorithms.

We now introduce two simple modifications to the scheme above that produce a more efficient dynamic programming algorithm. The new algorithm will be computationally tested in Section 7.

The first observation is that a similar recursion may be obtained that does not require the *explicit* storage of the two matrices. Indeed, at each stage j, only entries from column $j - 1$ are used to update the current stages, which allows for a reduction of the space requirement. If stages are updated in a proper order (namely, for capacity values from C down to 0), two vectors of size $C + 1$ are enough to store all A and B states (provided all input data are stored as well). When large instances are considered, this space reduction produces considerable improvements in terms of computing times too in practice. However, in this way one is no more able to detect the optimal solution vector. To recover the solution vector, one could adopt a turnaround similar to that used in [5] storing, for each state, both the solution value and the associated set of used classes. An optimal KPS solution can be thus computed a posteriori, solving a KP instance defined by the items in the classes that are used in the most profitable state. In this case too, one may expect some increase in both the storage requirement and in the computing time. As observed in [20], an optimal solution vector can be retrieved after the dynamic programming algorithm is completed, simply adapting the general recursive storage reduction principle from [19], thus preserving both the original running time and space complexity. The reader is referred to [13, Sec. 3.3] for a detailed description of this scheme.

6. A relevant special case

In this section we introduce a special relevant case that may be encountered when solving KPS. This happens if, for each class $i \in I$, the following condition is satisfied

$$s_i + \sum_{j \in K_i} w_j \leq C \tag{30}$$

This means that, for each class i, all items of the class can be allocated into the knapsack.

We observe that KPS remains NP-hard also in case assumption (30) is valid. Moreover, this special case is relevant from a theoretical viewpoint: while KPS does not admit a polynomial time approximation algorithm with a bounded approximation ratio, there exists an FPTAS that can be derived when assumption (30) is satisfied (see, [20]). In addition, this setting makes sense in case the setup capacities play a role in the definition of the problem; indeed, if (30) is not satisfied, it is likely that the optimal solution includes only items from a single class, as using additional classes would consume some more capacity in the knapsack. Finally, this situation is always satisfied for the instances in our testbed that are taken from the literature (see Section 6.1).

Therefore, for the rest of this section we will assume that (30) is valid.

Observation 4. *Under assumption (30) LP1 dominates LP2.*

Proof. Note that, for each class $i \in I$, all items in K_i can be inserted in the knapsack. Thus, we have $C_i^{\overline{w}} = \sum_{j \in K_i} \overline{w}_j$, i.e., the surrogate capacity can be computed in linear time. In order to show the result, we have to prove that every feasible solution to LP1 is feasible for LP2. This can be trivially proved as, for each class i, the surrogate constraint (9) in M2 can be obtained summing up constraints (4) associated with items $j \in K_i$ using non-negative coefficients $\overline{w}_j$. To conclude the proof, one can observe that instances exist which are feasible for LP2 but not for LP1, see for example the class of instances described in the second part of Observation 1. □

Observation 5. *Under assumption (30) LP3 can be computed in $O(n)$ time.*

Proof. Consider a given class i. If all items in K_i can be inserted in the knapsack, the pricing problem (26) for a given λ^* has the following optimal solution:

$$\theta_j \;=\; \begin{cases} 1 & \text{if } p_j - \lambda^* w_j > 0 \\ 0 & \text{otherwise} \end{cases} \qquad (j \in K_i)$$

Since items are sorted according to non-increasing profit over weight ratio, this means that all items $j \in [\alpha_i, \gamma_i(\lambda^*)]$ will be selected, where $\gamma_i(\lambda^*) = \min\{j \in K_i : p_j/w_j \leq \lambda^*\}$. Thus, at most n_i variables associated with class i have to be considered into the model—namely, for each item $j \in K_i$, one variable corresponding to item set $[\alpha_i, j]$. Overall, model M3 is thus an MCKP with n variables, whose LP relaxation can be solved in $O(n)$ time using the algorithm presented by [9] and [22]. □

Observation 6. *Under assumption (30) LP1 has the same upper bound as LP3.*

Proof. We already proved in Observation 2 that every feasible solution to LP3 can be converted into a feasible solution to LP1, and hence the latter cannot be better (i.e., lower) that the former. Thus, we only have to show that any optimal solution, say x^*, y^*, to LP1 corresponds to a feasible solution for LP3 with the same value. The algorithm depicted in Figure 1 shows that, for each class i, the solution has the following form: either

(a) $x_j^* = y_i^* \quad \forall j \in [\alpha_i, \gamma_i]$ for some γ_i; or

(b) $x_j^* = 1 \quad \forall j \in [\alpha_i, \gamma_i]$ and $x_j^* = 0 \quad \forall j \in [\gamma_i + 2, \beta_i]$ for some γ_i.

Case (a) may arise when all items of the class have been either fully packed or not packed at all, or if the critical item corresponds to the cumulative item for the class; in this case, all the associated items take the same fractional value. Case (b) happens when the critical item is an original item, say $\gamma_i + 1$, which can be inserted only after all preceding items have been completely packed; in this case, only this item is packed in a fractional way and next items are disregarded. Given x^*, y^* let us define, for each class i, item sets $S_1^i := \{\alpha_i, \gamma_i\}$ and $S_2^i := \{\alpha_i, \gamma_i + 1\}$. Since both satisfy the capacity constraint, we can introduce the associated variables in M3, that will be denoted by $\xi_{S_1}^i$ and $\xi_{S_2}^i$, respectively. Now, a feasible solution to LP3 is obtained by setting, for each class i:

- $\xi_{S_1}^i = y_i^*$ and $\xi_{S_2}^i = 0$, in case (a); and
- $\xi_{S_1}^i = 1 - \theta^*$ and $\xi_{S_2}^i = \theta^*$, in case (b),

where $\theta^* = x_{\gamma_i+1}^*$ is the value of the critical item in the optimal solution to LP1. □

6.1. Instances from the literature

To the best of our knowledge, only few sets of test instances have been proposed in the literature for knapsack problems with setup. The 180 randomly generated instances proposed by [18] are not publicly available and refer to KPS with additional upper bounds on the maximum weight that can be used for each item class. We generated this testbed of instances following the description of [18]. Thanks to the impressive improvements of commercial ILP solvers in the last decade, all these instances are now easily solved directly using a general-purpose ILP solver on model M1.

Much harder KPS instances have been proposed in [5]. These problems, publicly available at `https://sites.google.com/site/chebilkh/knapsack-problem-with-setup`, were generated to simulate realistic instances from an industrial application. In particular, these instances have been randomly generated with a number of items $n \in \{500, 1000, 2500, 5000, 10000\}$ and a number of classes $m \in \{5, 10, 20, 30\}$; ten instances have been generated for each pair (n, m), thus producing a testbed of 200 problems. Item profits and weights have been generated so as to have strongly correlated instances, and the setup cost (resp. capacity) of each class is a randomly number correlated to sum of the profits (resp. weights) of the items in the class. Observe that this benchmark has also been used by other recent works on KPS (see the next Section), and that all instances in this set satisfy condition (30).

Recently, four additional classes of instances have been introduced in [7]. The authors kindly provided us all these instances in a private communication. These instances, that were generated according to the scheme given in [21], have been denoted as Classes 1, 2, 3 and 5, whereas Class 4 is used to refer to the problems proposed by [5]. Classes 1, 2 and 3 refer to uncorrelated (w_j

and p_j in $[10, 10000]$), correlated (w_j in $[10, 10000]$ and p_j in $[w_j - 1000, w_j + 1000]$) and strongly correlated (w_j in $[10, 100]$ and $p_j = w_j + 10$) instances, respectively. Each class contains 160 problems. For these instances, the number of item classes is $m \in \{50, 100\}$, while the number of items n_i for each class $i \in I$ is uniformly distributed in the ranges $[40, 60]$ and $[90, 110]$. The setup cost f_i and setup capacity s_i are defined accordingly to the following formulas: $f_i = e_1 \left(\sum_{j \in K_i} p_j\right)$ and $d_i = e_2 \left(\sum_{j \in K_i} w_j\right)$, where e_1 and e_2 are uniformly distributed in the intervals $[0.05, 0.15]$, $[0.15, 0.25]$, $[0.25, 0.35]$ and $[0.35, 0.45]$. Finally, Class 5 includes 100 larger strongly correlated instances with up to 100000 items and 200 classes, where setup costs and capacities are obtained taking $e1 = e2$ uniformly ranging in the interval $[0.15, 0.25]$. For all classes proposed in [7], the capacity C is an integer value randomly generated in the range $[0.4(\sum_{j \in N} w_j), 0.6(\sum_{j \in N} w_j)]$. It turns out that all the instances in the new benchmark satisfy property (30).

7. Computational Experiments

In this section we perform an extensive computational analysis on the performances of our approaches using the benchmarks from the literature described in Section 6.1 and new randomly generated instances. All algorithms were implemented using C and were run on an Intel Xeon E3-1220 V2 running at 3.10 GHz in single-thread mode using `IBM-ILOG` Cplex 12.6 (`CPLEX` in the following) as ILP solver.

7.1. Solving the LP relaxation of the models

Our first set of experiments is aimed at evaluating the computational effort required to compute the LP relaxation of the models and the quality of the associated upper bound. For this purpose, we considered the instances proposed in [5] only. Table 1 reports, for each model, the computing time (in seconds) needed to compute the relaxation using `CPLEX`, and the associated percentage gap, computed as $\%gap = 100 * \frac{U - z^*}{z^*}$, where U and z^* denote the value of the relaxation and of the optimal solution, respectively. Columns RLP2_A and RLP2_B correspond to the LP relaxations of models M2_A and M2_B, respectively (see Section 3.2). All figures report average values over the 10 instances having the same values for m and n.

These results confirm the theoretical dominance among the relaxations, as shown in Observation 4: LP1 provides a very tight upper bound on the optimal value, while RLP2 usually yields a poor approximation. However, the computing time required to `CPLEX` for solving the latter, in both the versions addressed, is considerably smaller than for the computation of the former relaxation. Using the combinatorial algorithms described in Sections 3.1 and 3.2, the computing time required to solve the relaxations above is always negligible.

7.2. Exact solution of the models M1 and M2 using `CPLEX`

In our second set of experiments, we solved the instances proposed in [5] using `CPLEX` on models M1 and M2, allowing for a time limit of 3600 seconds per instance. As already experienced in many other papers from the literature (see, e.g., [5] and [7]), even using a state-of-the-art MIP solver with these model yields poor results in practice. In particular, using the models above, `CPLEX`

Instances		LP1		RLP2_A		RLP2_B	
m	n	Time (sec.)	% gap	Time (sec.)	% gap	Time (sec.)	% gap
5	500	0.010	1.542	0.009	10.914	0.013	5.911
	1000	0.016	1.322	0.000	12.136	0.000	10.405
	2500	0.051	0.762	0.000	10.534	0.000	5.520
	5000	0.121	0.708	0.010	11.730	0.010	10.022
	10000	0.362	0.386	0.010	10.217	0.010	5.256
10	500	0.010	0.961	0.000	9.956	0.000	4.970
	1000	0.011	2.126	0.000	11.391	0.000	6.268
	2500	0.040	2.223	0.010	11.784	0.002	6.727
	5000	0.093	0.373	0.010	11.298	0.010	9.568
	10000	0.418	0.309	0.010	11.356	0.010	9.627
20	500	0.007	0.130	0.000	6.205	0.000	2.146
	1000	0.010	0.487	0.000	9.065	0.000	4.369
	2500	0.035	0.690	0.010	11.244	0.010	9.463
	5000	0.074	0.356	0.010	11.089	0.010	9.355
	10000	0.253	0.302	0.010	11.191	0.010	9.493
30	500	0.007	0.098	0.000	5.965	0.000	1.999
	1000	0.010	0.381	0.000	8.641	0.000	3.981
	2500	0.031	0.284	0.010	9.475	0.010	4.854
	5000	0.063	0.089	0.010	10.709	0.010	8.994
	10000	0.163	0.090	0.010	10.936	0.010	9.158

Table 1: LP relaxation of the models, solved using CPLEX

was unable to solve instances with more than 2500 items in a systematic way. Surprisingly, model M1, though having the tightest LP relaxation, is the least effective (among the models that are compared) when integrality is required, in terms of number of instances solved to proven optimality and average computing time, whereas model M2_B, is able to solve 65% of the instances with an average computing time of 5 minutes.

7.3. Combinatorial algorithms

In this section we evaluate the computational performances of our exact approaches for KPS. In particular, we consider B&B$_{\mathrm{LP1}}$ and B&P$_{\mathrm{LP3}}$, that denote the Branch-and-Bound and Branch-and-Price algorithms based on the LP relaxations of models M1 (strengthened as shown in Section 3.1) and M3, respectively, and the improved dynamic programming algorithm (DP in the following) described in Section 5. Preliminary computational experiments showed that the branch-and-bound algorithm based on model M2 is dominated by the other approaches; hence, we do not report results for this algorithm.

We used the same benchmark considered in the previous sections, and compared our algorithms with the exact algorithms given in [7] and in [20]. In the following, these algorithms will be denoted as DSS and PS, respectively. Both algorithms showed to outperform the dynamic programming in [5] on our benchmark; for this reason, the latter is excluded from comparison. In a personal communication, the authors of [7] provided us with the implementation of their algorithm, which

allows a better performance evaluation with our new exact algorithms. Since all our algorithms but DSS are sequential, in a first round of tests we ran DSS on our machine using 1 thread only (see the next section for a multi-thread performance analysis). We observe that the results for algorithm PS are taken from [20] and were obtained on an Intel i5 CPU running at 3.2 GHz with 16 GB of RAM.

Table 2 reports the performance comparison of the sequential algorithms. Each line of the table refers to 10 instances with the same number of classes and items. For each algorithm we report the average and maximum computing time (in seconds) for solving the associated instances.

Instances		B&B$_{\text{LP1}}$		B&P$_{\text{LP3}}$		DP		DSS		PS	
		Time (sec.)		Time (sec.)		Time (sec.)		Time (sec.)		Time (sec.)	
m	n	avg	max	avg	max	avg	max	avg	max	avg	max
5	500	0.00	0.00	0.00	0.00	0.02	0.02	0.08	0.11	0.02	0.02
	1000	0.00	0.00	0.00	0.01	0.06	0.06	0.13	0.15	0.07	0.08
	2500	0.00	0.01	0.01	0.01	0.25	0.25	0.35	0.38	0.54	0.56
	5000	0.00	0.01	0.01	0.01	0.90	0.91	0.94	1.07	2.00	2.16
	10000	0.01	0.01	0.01	0.02	3.66	3.69	3.50	4.17	8.67	8.98
10	500	0.00	0.01	0.00	0.01	0.02	0.02	0.13	0.19	0.01	0.02
	1000	0.01	0.01	0.01	0.01	0.06	0.07	0.14	0.18	0.07	0.08
	2500	0.02	0.02	0.01	0.02	0.25	0.25	0.35	0.39	0.49	0.50
	5000	0.01	0.02	0.01	0.02	0.93	0.93	0.86	0.92	1.79	1.84
	10000	0.03	0.03	0.02	0.03	3.70	3.76	2.76	3.01	7.15	7.33
20	500	0.00	0.00	0.00	0.00	0.02	0.02	0.12	0.17	0.02	0.02
	1000	0.20	0.53	0.15	0.38	0.06	0.06	0.27	0.48	0.07	0.07
	2500	3.29	6.18	1.41	2.72	0.25	0.26	0.45	0.76	0.42	0.45
	5000	3.23	8.93	1.23	3.22	0.94	0.95	0.87	1.16	1.68	1.70
	10000	7.53	15.60	2.61	5.16	3.76	3.81	2.64	2.88	6.69	6.71
30	500	0.00	0.02	0.01	0.03	0.02	0.02	0.36	1.20	0.02	0.03
	1000	3.09	6.64	2.68	5.56	0.06	0.06	0.87	2.77	0.06	0.07
	2500	19.41	72.34	9.98	37.84	0.25	0.26	0.53	0.66	0.45	0.46
	5000	1.44	8.17	0.66	3.67	0.94	0.95	1.02	1.59	1.66	1.68
	10000	12.79	84.94	4.85	30.69	3.76	3.77	2.92	4.44	6.58	6.74

Table 2: Sequential algorithms for the exact solution of KPS.

Results in Table 2 show that our enumerative algorithms are competitive with both DSS and PS on this benchmark. There are some specific sets of problems in which they are considerably faster than the previous approaches from the literature, in particular when the number of item classes is "small", regardless of the number of items. We also observe that M3 is typically faster than M1. Indeed, as condition (30) is always verified on these instances, these two algorithms explore a similar number of nodes. However, the computation of LP3 is usually faster than that of LP1 due to two main factors: first, M3 is usually able to terminate the enumeration generating only few variables, hence the model maintains a small size. In addition, the *practical* complexity of computing LP3 is usually easier than the *theoretical* one, especially when some y variables have been fixed to zero by branching. In this case, M3 simply disregardes the associated classes, thus saving computing time, while the algorithm for computing LP1 has to scan in any case the associated items, though inserting them in the solution is forbidden. Our improved dynamic programming DP has good performances

when the number of items is small, regardless the number of classes. Given its $O(nC)$ complexity, the required computing time is strongly dependent on the number of items. A similar behaviour can be observed evaluating the results of the dynamic programming algorithm `PS` proposed in [20]. Finally, we note that both M1 and M3 may have large variability for what concerns the solution time, which is not the case for our dynamic programming algorithm DP.

7.4. Parallel algorithms

Our last set of experiments addresses the more challenging instances recently proposed in [7]. As algorithm `DSS` is based on an ILP solver and may take advantage of the availability of a multi-thread architecture, we ran this code on our machine allowing for 4 cores (as in [7]). Observe that in our tests we used `IBM-ILOG` Cplex 12.6, that is a more recent release than the 12.5 one used in [7]. For this reason, computing times reported in Tables 3–5 for algorithm `DSS` may be different (typically, smaller) than those reported in [7].

To have a fair comparison, we implemented a parallel variant of our algorithm (FMT in the following) that executes algorithms B&B$_{\mathrm{LP1}}$, B&P$_{\mathrm{LP3}}$ and DP in parallel, and halts execution as soon as one of the 3 terminates.

Tables 3, 4 and 5 report the associated results for the all the classes of instances considered in [7]. Each line of these tables gives the average and the maximum computing time over ten instances of similar characteristics.

Instances			Class 1				Class 2				Class 3			
			FMT		`DSS`		FMT		`DSS`		FMT		`DSS`	
			Time (sec.)		Time (sec.)		Time (sec.)		Time (sec.)		Time (sec.)		Time (sec.)	
m	n_i	setup	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max
50	[40-60]	[0.05-0.15]	0.00	0.00	0.24	0.31	0.01	0.03	0.30	0.45	0.18	0.32	0.78	1.95
		[0.15-0.25]	0.00	0.00	0.27	0.34	0.01	0.04	0.31	0.45	0.17	0.32	0.71	1.32
		[0.25-0.35]	0.00	0.00	0.23	0.29	0.03	0.07	0.36	0.51	0.16	0.30	0.49	1.00
		[0.35-0.45]	0.00	0.00	0.23	0.32	0.00	0.01	0.35	0.50	0.11	0.29	0.95	2.80
50	[90-110]	[0.05-0.15]	0.00	0.00	0.63	0.72	0.07	0.25	0.87	1.11	0.52	1.28	1.23	3.30
		[0.15-0.25]	0.00	0.00	0.64	0.80	0.49	4.26	0.79	0.95	0.52	1.23	1.15	2.52
		[0.25-0.35]	0.00	0.01	0.57	0.80	2.33	11.80	0.90	1.32	0.44	1.14	0.79	1.03
		[0.35-0.45]	0.00	0.00	0.43	0.60	0.01	0.05	0.67	0.93	0.92	1.26	1.11	1.82
100	[40-60]	[0.05-0.15]	0.00	0.00	0.53	0.61	0.06	0.16	0.64	0.84	0.55	1.24	2.09	7.63
		[0.15-0.25]	0.00	0.01	0.51	0.66	0.02	0.05	0.70	0.90	0.60	1.04	1.24	3.14
		[0.25-0.35]	0.00	0.01	0.50	0.65	0.11	0.43	0.79	1.09	0.83	1.21	1.46	3.58
		[0.35-0.45]	0.00	0.00	0.43	0.56	0.01	0.01	0.54	0.69	0.61	1.22	4.40	26.75
100	[90-110]	[0.05-0.15]	0.00	0.01	1.30	1.51	1.82	6.23	1.46	1.82	2.76	4.90	2.56	4.49
		[0.15-0.25]	0.01	0.01	1.15	1.35	4.56	41.18	1.67	3.11	2.92	4.61	2.64	7.39
		[0.25-0.35]	0.01	0.02	1.11	1.69	4.93	24.00	1.57	2.49	2.14	4.43	2.26	4.61
		[0.35-0.45]	0.01	0.01	0.87	1.15	0.22	1.61	1.48	1.95	2.75	4.26	2.73	5.70

Table 3: Parallel algorithms on instances of Classes 1, 2 and 3.

In Table 3 we compare the performances of FMT and DSS for the first three classes, that include 480 instances obtained using the same instance generator parameters for m, n_i and setup. While $m \in \{50, 100\}$ is the number of classes, n_i represents the interval used to determine the number of items for each class $i \in I$ and column "setup" gives the interval used for parameters e_1 and e_2, see Section 6.1. These results show that our algorithm FMT is competitive with algorithm DSS: for instances of Class 1, FMT outperforms DSS, often by orders of magnitude. Overall these 480 instances, the average computing times of the algorithms are 0.63 and 1.03 respectively, and FMT is faster than DSS in 431 cases.

Table 4 reports the results on instances of Class 4, i.e., the benchmark proposed in [5] that was already considered in Table 2. On these instances, algorithm FMT is typically faster that DSS, with a considerable speedup for instances with $m \leq 10$. Observe that, in some cases, algorithm DSS takes fully advantage of the paralelization and reaches a speedup that is even larger than the number of threads that are used.

Instances		FMT		DSS	
		Time (sec.)		Time (sec.)	
m	n	avg	max	avg	max
5	500	0.00	0.00	0.09	0.11
	1000	0.00	0.00	0.14	0.17
	2500	0.00	0.01	0.36	0.37
	5000	0.01	0.01	0.48	0.50
	10000	0.01	0.01	0.84	0.92
10	500	0.00	0.01	0.14	0.20
	1000	0.01	0.01	0.15	0.19
	2500	0.01	0.02	0.34	0.39
	5000	0.01	0.02	0.53	0.56
	10000	0.02	0.03	0.90	0.97
20	500	0.00	0.00	0.14	0.20
	1000	0.06	0.06	0.25	0.42
	2500	0.23	0.26	0.44	0.74
	5000	0.68	0.99	0.65	0.73
	10000	2.43	3.88	1.01	1.11
30	500	0.00	0.02	0.40	1.29
	1000	0.06	0.06	0.53	1.40
	2500	0.23	0.26	0.49	0.59
	5000	0.36	0.98	0.79	1.05
	10000	1.66	3.88	1.51	2.34

Table 4: Parallel algorithms on instances of Class 4.

Results on instances of Class 5 are given in Table 5. On this benchmark, algorithm DSS performs slightly better than FMT on average. However, while the computing time of FMT seems to be highly dependent on the number of classes, the performances of DSS are highly influenced by the number of items. For this reason, our approach turns out to be much faster than DSS for all problems that have $m \leq 10$, independently on the number of items. This is not surprising, as our approach was originally designed to solve the instances proposed in [5], that have a small number of

classes, and exploits the branching on the y variables. On the other hand, algorithm DSS is based on the application of a general purpose ILP solver, which may get into troubles when the number of decision variables gets too large.

Instances		FMT		DSS	
		Time (sec.)		Time (sec.)	
m	n	avg	max	avg	max
5	20000	0.01	0.01	1.59	1.96
	50000	0.02	0.04	6.44	9.34
	100000	0.04	0.12	42.97	48.25
10	20000	0.03	0.05	1.40	1.65
	50000	0.07	0.09	4.05	5.95
	100000	0.12	0.16	12.89	23.75
20	20000	4.34	10.45	1.79	2.05
	50000	9.53	23.03	5.20	5.95
	100000	17.71	44.24	15.65	23.28
30	20000	4.12	16.90	2.15	2.66
	50000	23.21	106.60	4.97	6.15
	100000	120.44	429.62	15.38	23.40
50	20000	7.47	17.18	2.09	3.19
	50000	53.30	107.74	6.44	7.96
	100000	85.27	431.74	15.98	19.75
100	20000	9.33	17.17	4.48	10.37
	50000	47.89	107.89	8.49	26.93
	100000	162.00	432.90	11.11	15.97
200	20000	12.52	17.34	6.40	17.01
	50000	58.63	108.76	14.69	33.55
	100000	224.22	434.94	15.62	27.25

Table 5: Parallel algorithms on instances of Class 5.

To better investigate the performances of FMT and DSS, we generated a new class of large instances, called Class 6 in the following. The item weights and profits (w_j and p_j, $j \in N$) were generated to define hard knapsack (KP) instances, using the generator proposed in [17] (available at `http://www.diku.dk/~pisinger/codes.html`). More in details, we generated instances belonging to the following eight classes of KP instances:

1. *Uncorrelated*: w_j u.r. in $[1, \overline{R}]$, p_j u.r. in $[1, \overline{R}]$.
2. *Weakly correlated*: w_j u.r. in $[1, \overline{R}]$, p_j u.r. in $[\max\{1, w_j - \overline{R}/10\}, w_j + \overline{R}/10]$.
3. *Strongly correlated*: w_j u.r. in $[1, \overline{R}]$, $p_j = w_j + \overline{R}/10$.
4. *Inverse strongly correlated*: p_j u.r. in $[1, \overline{R}]$, $w_j = p_j + \overline{R}/10$.
5. *Almost strongly correlated*: w_j u.r. in $[1, \overline{R}]$, p_j u.r. in $[w_j + \overline{R}/10 - \overline{R}/500, w_j + \overline{R}/10 + \overline{R}/500]$.
6. *Subset-sum*: w_j u.r. in $[1, \overline{R}]$, $p_j = w_j$.

7. *Even-odd subset-sum*: w_j even value u.r. in $[1, \overline{R}]$, $p_j = w_j$, C odd.

8. *Even-odd strongly correlated:* w_j even value u.r. in $[1, \overline{R}]$, $p_j = w_j + \overline{R}/10$, C odd.

where $\overline{R} = 1000$ and u.r. stands for "uniformly random integer".

In order to transform a KP instance into a KPS instance, we defined the knapsack capacity with a formula similar to the one used in [7], i.e., $C = e_3 \sum_{j \in N} w_j$ where $e_3 \in \{0.45, 0.5, 0.55\}$. The setup cost f_i and setup capacity s_i are also defined similarly to [7], setting $e_1 = e_2 = 0.05$. The items are uniformly partitioned among the classes, i.e., $|K_i| = \frac{n}{m}$ $(i \in I)$. In this way, for each size $n \in \{5000, 10000, 20000, 100000, 200000\}$ and $m \in \{5, 10\}$, we generated 24 instances. In total this new testbed is composed of 240 instances. All these instances as well satisfy property (30) discussed in Section 6. For these experiments, the time limit has been set to 1200 seconds per instance.

Each row of Table 6 gives the results for the 24 instances of Class 6 that have the same number of classes m and number of items n. For both FMT and DSS we report the number of instances solved to proven optimality within the time limit, and the average and maximum computing times.

Instances		FMT			DSS		
			Time (sec.)			Time (sec.)	
m	n	#opt	avg	max	#opt	avg	max
5	5000	24	0.00	0.00	22	101.34	1200.00
	10000	24	0.00	0.01	24	1.27	3.42
	20000	24	0.00	0.01	22	103.22	1200.00
	100000	24	0.02	0.06	24	19.07	60.57
	200000	24	0.03	0.07	24	73.87	287.91
10	5000	24	0.00	0.01	24	0.73	1.19
	10000	24	0.01	0.03	23	51.98	1200.00
	20000	24	0.02	0.06	23	53.01	1200.00
	100000	24	0.08	0.17	24	21.21	44.34
	200000	24	0.16	0.32	24	67.71	138.84

Table 6: Parallel algorithms on instances of Class 6.

These results confirm that for hard instances with a small number of classes our approach is much faster than DSS. The new FMT algorithm is able to solve each of the 240 instances in at most half a second, whereas algorithm DSS fails in solving 6 problems within the time limit and has an average computing time of about 50 seconds.

8. Conclusions

We considered a variant of the knapsack problem with setups associated to classes of items. We studied alternative ILP formulations and analysed their properties in terms of linear programming relaxation. We proposed a generic Branch-and-Bound framework capable of embedding different relaxations and we showed how to solve these relaxations via new combinatorial algorithms (one of

which is based on Column Generation). Finally, we proposed a parallel algorithm called FMT which combines the strengths of the new Branch-and-Bound algorithms and of an improved Dynamic Programming algorithm. We computationally compared the performances of the state-of-the-art algorithms for KPS with FMT. The outcome of these experiments is that FMT is capable of efficiently solving all the instances of the literature and it is the best algorithm for instances with a small number of classes.

Future lines of research. An important generalization of KPS arises when lower and upper bounds are imposed on the total weight of the selected items for each class (if used). While the case where an upper bound is imposed has been studied by [18], to the best of our knowledge the case with a lower bound has not been considered so far in the literature. A challenging topic in this area is thus the extension of our approaches to these new constraints, that may prevent the linear-time algorithms developed for LP1 and LP3 from being valid. In a similar way, the interaction of setup costs and different constraints, e.g., as precedences and/or incompatibilities among items, may be worth of studying.

9. Acknowledgments

The authors thank Federico Della Croce, Fabio Salassa and Rosario Scatamacchia for making the source code for their algorithms available, and for having provided detailed computational results for their algorithms of [7].

[1] U. Akinc. Approximate and exact algorithms for the fixed-charge knapsack problem. *European Journal of Operational Research*, 170(2):363–375, 2006.

[2] N. Altay, P. R. Jr., and K. Bretthauer. Exact and heuristic solution approaches for the mixed integer setup knapsack problem. *European Journal of Operational Research*, 190(3):598–609, 2008.

[3] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28(5):1130–1154, 1980.

[4] E. Chajakis and M. Guignard. Exact algorithms for the setup knapsack problem. *INFOR: Information Systems and Operational Research*, 32(3):124–142, 1994.

[5] K. Chebil and M. Khemakhem. A dynamic programming algorithm for the knapsack problem with setup. *Computers & Operations Research*, 64:40–50, 2015.

[6] G. Dantzig. Discrete variable extremum problems. *Operations Research*, 5(2):266–277, 1957.

[7] F. Della Croce, F. Salassa, and R. Scatamacchia. An exact approach for the 0–1 knapsack problem with setups. *Computers & Operations Research*, 80:61–67, 2017.

[8] J. Desrosiers and M. Lübbecke. *Column Generation*, chapter A Primer in Column Generation. Springer US, Boston, MA, 2005.

[9] M. Dyer. An o(n) algorithm for the multiple-choice knapsack linear program. *Mathematical Programming*, 29(1):57–63, 1984.

[10] M. Guignard. Solving makespan minimization problems with lagrangean decomposition. *Discrete Applied Mathematics*, 42(1):17–29, 1993.

[11] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21(2):277–292, 1974.

[12] E. Johnson and M. Padberg. A note of the knapsack problem with special ordered sets. *Operations Research Letters*, 1(1):18–22, 1981.

[13] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin Heidelberg, 2004.

[14] E. Lin. A bibliographical survey on some well-known non-standard knapsack problems. *INFOR*, 36(4):274–317, 1998.

[15] S. Martello and P. Toth. An upper bound for the zero-one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1(3):169–175, 1977.

[16] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, New York, 1990.

[17] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

[18] S. Michel, N. Perrot, and F. Vanderbeck. Knapsack problems with setups. *European Journal of Operational Research*, 196(3):909–918, 2009.

[19] U. Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63(4):419–430, 1999.

[20] U. Pferschy and R. Scatamacchia. Improved dynamic programming and approximation results for the knapsack problem with setups. *Optimization Online*, 2017.

[21] Y. Yang and R. Bulfin. An exact algorithm for the knapsack problem with setup. *International Journal of Operational Research*, 5(3):280–291, 2009.

[22] E. Zemel. An o(n) algorithm for the linear multiple choice knapsack problem and related problems. *Information Processing Letters*, 18(3):123–128, 1984.

Appendix: Solving the LP relaxation of M1

Determine the break item

Figure 4 reports the pseudo-code of the algorithm that can be used to determine the break item for a given item class in linear time.

Algorithm *Break Item*:

// initialization
set $J_0 = J_1 = \emptyset$, $J_B = \{\alpha_i, \ldots, \beta_i\}$, $P_1 = W_1 = 0$, $partition = \texttt{false}$;
// iterative steps
while $partition = \texttt{false}$ **do**
  find the median λ of the values in $R = \{\frac{p_j}{w_j} : j \in J_B\}$;
  $G := \{j \in J_B : \frac{p_j}{w_j} > \lambda\}$; $L := \{j \in J_B : \frac{p_j}{w_j} < \lambda\}$; $E := \{j \in J_B : \frac{p_j}{w_j} = \lambda\}$;
  $\lambda_{\max} := 0$
  if $|L| > 0$ **then**
    $\lambda_{\max} := \max\{\frac{p_j}{w_j} : j \in L\}$;
  endif;
  $\Phi_G := \frac{P_1 + \sum_{j \in G} p_j - f_i}{W_1 + \sum_{j \in G} w_j + s_i}$; $\Phi_{G \cup E} := \frac{P_1 + \sum_{j \in G \cup E} p_j - f_i}{W_1 + \sum_{j \in G \cup E} w_j + s_i}$;
  if $\Phi_G \leq \lambda$ and $\Phi_{G \cup E} > \lambda_{\max}$ **then**
    $partition = \texttt{true}$
  else
    if $\Phi_G > \lambda$ **then** // λ is too small (too many items precede the break item)
      $J_0 = J_0 \cup L \cup E$; $J_B = G$;
    else
      if $\Phi_{G \cup E} \leq \lambda_{\max}$ **then** // λ is too large (too few items precede the break item)
        $J_1 = J_1 \cup G \cup E$; $J_B = L$; $P_1 = P_1 + \sum_{j \in G \cup E} p_j$; $W_1 = W_1 + \sum_{j \in G \cup E} w_j$;
      endif;
    endif;
  endif;
end while;
$J_1 = J_1 \cup L$; $J_0 = J_0 \cup G$; $J_B = E(= \{e_1, \ldots, e_q\})$; $\sigma = \min\{j : \frac{P_1 + \sum_{i=1}^{j} p_{e_i} - f_i}{W_1 + \sum_{i=1}^{j} w_{e_i} + s_i} \geq \lambda\}$;
return e_σ.

Figure 4: Algorithm to find the break item b_i of a given class i, $i \in I$.

Proof of Theorem 2

Proof. Consider a class $i \in I$ and let $z_i^* = \max\{x_j^* : j \in K_i\}$ denote the maximum value for an x variable in the class.

Property 1. states that variable y_i^* must be at its lowest possible value z_i^* in any optimal solution. Otherwise, due to constraints (4), we must have $y_i^* > z_i^*$; in this case, however, reducing the value of y_i^* to z_i^* produces a solution which is still feasible and whose profit is not smaller than the original one.

Property 2. indicates that all variables associated with the first items (up to the break item) always attain the maximum value. By contradiction, assume this property is violated, and let $j = \min\{k \in [\alpha_i, b_i] : x_k^* < z_i^*\}$ and $h = \max\{k \in K_i : x_k^* = z_i^*\}$ denote the first item that has $x_j^* \neq z_i^*$ and the last item with $x_h^* = z_i^*$, respectively. Note that by definition item j has $x_j^* < z_i^*$, whereas Property 1. ensures that item h exists.
If $h > j$ a simple swap argument shows that reducing x_h to release some capacity and increasing x_j to fill this capacity yields a feasible solution whose profit is not smaller than the original one. After this operation one may be required to redefine the correct value for y_i^*; the swap argument can be repeated, possibly redefining item h, until x_j hits the current y_i^* value.
In case $h < j$ it must be $h = j - 1$ (by definition of j). Observing that $\frac{P(h)}{W(h)} \leq \frac{p_j}{w_j}$ a similar swap argument can be applied. A feasible solution can be obtained (i) reducing both variables y_i and variables x_j associated with items $\alpha_i, \ldots, j-1$ by some positive ϵ, thus freeing a capacity equal to $\epsilon W(h)$; and (ii) increasing the value of variable j by $\Delta_j = \epsilon \frac{W(h)}{w_j}$. This new solution has at least the same profit as the initial solution, and is feasible for all values of ϵ such that $x_j^* + \Delta_j \leq y_i^* - \epsilon$, i.e., $0 < \epsilon \leq \frac{y_i^* - x_j^*}{1 + \frac{W(h)}{w_j}}$. □