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 effectively solve the problem to proven optimality. 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 $N = \{1, \ldots, n\}$ 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
exceed the capacity. KP can be formulated using the following Integer Linear Program (ILP):

$$\max \left\{ \sum_{j \in N} p_j x_j : \sum_{j \in N} w_j x_j \leq C, x_j \in \{0, 1\}, j \in N \right\}$$ (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 [17, 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 positive setup cost $f_i$ is incurred and a positive 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 [15] 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 presents an exact algorithm based on a branch-and-bound scheme, while the latter uses cross decomposition to solve the case in which items can be taken at a fractional level. The problem addressed by [19] 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 [22]. 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 [21] and [6]. 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 turn out to be easy to solve in practice. Computational experiments reported in [6], 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. For what concerns approximate solutions, we mention a recent paper [14], where a tree search combination heuristic is
presented. The algorithm is based on the definition of a truncated tree search, where at each level only potentially good nodes are candidates for further exploration, and is tested on the instances introduced in [5].

**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 two 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 LP 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 obtain the best computational performance, we propose a parallel algorithm which exploits the qualities of all the new exact algorithms. We test 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 \( i \in I \) and \( m > 1 \); otherwise, one could associate the setup capacity and cost to the items, yielding a KP. 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 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 \( w_j + s_j \leq C \); 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* and reported in Figure 1. The optimal

---

**Figure 1: Example 1**

- \( m = 2, n = 4, C = 152 \);
- \( n_1 = 2, \alpha_1 = 1, \beta_1 = 2, n_2 = 2, \alpha_2 = 3, \beta_2 = 4 \);
- \( f_1 = 10, s_1 = 10, f_2 = 9, s_2 = 6 \);
- \( p_1 = 84, w_1 = 75, p_2 = 75, w_2 = 72, p_3 = 70, w_3 = 64, p_4 = 71, w_4 = 78 \).
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 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) using 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 $y_i$ associated with item classes: in particular, each variable $y_i$ takes value 1 if and only if some item of class $i$ is included in the solution. The resulting model is as follows

\[
\max \sum_{j \in N} p_j x_j - \sum_{i \in I} f_i y_i \tag{2}
\]

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

\[
x_j \leq y_{t_j} \quad j \in N \tag{4}
\]

\[
x_j \in \{0, 1\} \quad j \in N \tag{5}
\]

\[
y_i \in \{0, 1\} \quad i \in I. \tag{6}
\]

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 the sum of the item weights and the class setups must not exceed the capacity. Inequalities (4) force a class to be used whenever some item of the class is selected. Finally, (5)–(6) impose all variables to be binary. It is worth mentioning that constraints (4)-(5) and the objective function force the $y_i$ 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:

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

\[
y_i \in [0, 1] \quad i \in I \tag{8}
\]
we obtain the LP relaxation of M1. We will denote by LP1 this relaxation and by $U^1$ the associated upper bound value. 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

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

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

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

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:

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

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

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

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). In addition, let $U^2_A$ and $U^2_B$ be the upper bound values associated with the LP relaxations of M2.A and M2.B, respectively. The following result shows that there is no dominance between the bounds given by the linear programming relaxations of the models above and LP1.
Proposition 1. There is no dominance between bounds $U_1^*, U_2^A$ and $U_2^B$.

Proof. We show the thesis by giving two numerical instances showing that there is no dominance between the three bounds. 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 $U_1^* = 144.254$. Model M2_A has $C_1^{\infty} = 75$ and $C_2^{\infty} = 142$. 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 $U_2^A = 140.183$. Model M2_B has $C_1^{\infty} = 1$ and $C_2^{\infty} = 2$, and an optimal solution of its LP relaxation is $x_1^* = x_3^* = 1$ and $y_1^* = 1, y_2^* = 0.5$, with value $U_2^B = 139.5$.

Conversely, consider the following instance with $m = 2$ classes. Class 1 has $n_1 = 2, f_1 = 20$ and $s_1 = 1$, while class 2 has $n_2 = 2, f_2 = 1$ and $s_2 = 1$, and the capacity is 42. The first item has $p_1 = 100$ and $w_1 = 30$, while all the remaining 3 items have profit 10 and weight 10. An optimal solution for LP1 is given by $x_1^* = x_2^* = y_1^* = 1$ and $x_3^* = x_4^* = y_2^* = 0.047$, and has value $U_1^* = 90.904$. Model M2_A has $C_1^{\infty} = 40$ and $C_2^{\infty} = 20$. An optimal solution of its LP relaxation is $x_1^* = x_3^* = 1, x_4^* = 0.071$ and $y_1^* = 0.75$ and $y_2^* = 0.535$, yielding an upper bound $U_2^A = 95.178$. Model M2_B has $C_1^{\infty} = C_2^{\infty} = 2$, and an optimal solution of its LP relaxation is $x_1^* = x_3^* = 1, x_4^* = 0.095$ and $y_1^* = 0.5, y_2^* = 0.547$, with value $U_2^B = 100.404$.

Observe that the result above is valid in case coefficients $C_i^{\infty}$ 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 $\mathcal{S}_i (i \in I)$ of feasible subsets of items $S \subseteq K_i$ satisfying the knapsack capacity $C$

$$\mathcal{S}_i = \left\{ S \subseteq K_i : \sum_{j \in S} w_j \leq C - s_i \right\}. \tag{13}$$

For each item subset $S \in \mathcal{S}_i$, 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 $S \in \mathcal{S}_i (i \in I)$, a binary variable $\xi_S$ which takes value 1 if and only if subset $S$ is included in the solution:
\[
\max \sum_{i \in I} \sum_{S \in \mathcal{A}_i} P_S \xi_S \tag{14}
\]
\[
\sum_{i \in I} \sum_{S \in \mathcal{A}_i} W_S \xi_S \leq C \tag{15}
\]
\[
\sum_{S \in \mathcal{A}_i} \xi_S \leq 1 \quad i \in I \tag{16}
\]
\[
\xi_S \in \{0, 1\} \quad i \in I, S \in \mathcal{A}_i. \tag{17}
\]

Objective function (14) maximizes the total profit of the selected subset of items, whereas constraint (15) ensures that the solution satisfies the capacity constraint. Inequalities (16) impose that at most one subset is selected for each class, whereas constraints (17) 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 (17) with the following ones:
\[
\xi_S \geq 0 \quad i \in I, S \in \mathcal{A}_i, \tag{18}
\]
we obtain the linear programming relaxation of M3, that will be denoted as LP3 in what follows. Let \(U^3\) be the associated upper bound value. Note that constraints (16) implicitly provide an upper bound of value 1 on the \(\xi_S\) variables, thus we do not need to impose this bound in (18).

Finally, we observe that the model above has already been proposed by [4] and used, e.g., by [19] 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 \(U^3\) cannot be worse than its counterpart associated with all the other models.

**Proposition 2.** Bound \(U^3\) dominates bounds \(U^1, U^2_A \) and \(U^2_B\).

**Proof.** To prove the result, it is enough to consider the model, say \(M^1+\), obtained by adding to \(M1\) the following valid constraints:
\[
\sum_{j \in K_i} w_j x_j \leq C - s_i y_i, \quad i \in I. \tag{19}
\]

We first observe that the these inequalities are implied by (3), hence their addition does not change the feasible region of the LP relaxation of \(M1^+\) with respect to that of \(M1\). We now apply Dantzig Wolfe Reformulation (DWR) to \(M1^+\) (see [7]) through a convexification of constraints (4) and (19). This is equivalent to impose the LP relaxation of such reformulation to be equal to the intersection of constraints (3) with the convex hull of constraints (4), (5), (6) and (19). The resulting model is exactly M3. One of the basic properties of DWR is that the LP relaxation of the reformulated
model is as strong as the original model. Moreover, since the convexified region does not have the integrality property (i.e., the constraints (19) strictly contain their convex hull), this fact implies that there exists at least one objective function for which the reformulation of $M1^+$ (i.e., $M3$) has a strictly better LP relaxation value than $M1$. A similar reasoning applies to models $M2_A$ and $M2_B$, i.e., after adding constraints (19), $M3$ can be obtained via a convexification of constraints (9) and (19).

For the sake of completeness, we consider again the instance of Example 1. In this case, 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 $U_3 = 135.482$, i.e., it is lower than $U_1, U_2^A$ and $U_2^B$, which are $144.254$, $140.183$ and $139.5$, respectively (see Proposition 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 a 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 [22] introduces some properties of an optimal solution to $LP1$ and derives a combinatorial algorithm for its solution. However, no detailed description nor analysis on the computational complexity of the algorithm is given in [22]. Thus, for the sake of completeness, we report a pseudo-code of the algorithm in Figure 2. In addition, we show that the algorithm can be executed in linear time, thus improving the time complexity over the result claimed in [1]. Finally, we propose a strengthened relaxation that can be computed in constant time if an optimal solution to $LP1$ is known.

In this algorithm, one has to compute, for each class $i$, the \textit{break item} $b_i$ and the associated \textit{macro-item} $I_i$. The former is the first item of the class for which the ratio between the cumulative profit and the cumulative weight is larger than the profit over weight ratio of all subsequent items (if no such item exists, the break item is the last item of the class), while the latter is an artificial item that simulates the profit and weight incurred when taking all items $j \in [\alpha_i, b_i]$ at the same value. For this reason, its profit (resp. weight) is equal to the sum of the profits (resp. weights) of all items $j \in [\alpha_i, b_i]$, plus the cost and capacity required for using the class. Given these figures, the algorithm adopts the well-known strategy for computing an optimal solution to the LP relaxation of KP, i.e., it takes one (either original or macro) item at a time until the entire capacity is used. The last selected item (or macro item) can then be taken at a fractional value. Finally, an optimal solution to $LP1$ is derived, observing that, for each class $i$, taking the associated macro-item at a given value, say $\theta_i$, corresponds to taking at the same value both $y_i$ and all $x_j$ variables associated with items $\in [\alpha_i, b_i]$.

In an optimal $LP1$ solution, though many items may be taken at a fractional value, at most one $y$ variable may be fractional. A similarity of $LP1$ with the LP relaxation of KP is given by the
Algorithm LP1:

initialize: \( N := \emptyset \);

for each class \( i \in I \) do
    compute the break item \( b_i \);
    define the macro-item \( I_i \);
    \( N := N \cup \{ I_i \} \cup \{ b_i + 1, \ldots, \beta_i \} \);
end do

Solve the LP relaxation of the KP instance defined by item set \( N \), and let \( \theta \) be the associated solution;

for each class \( i \in I \) do
    set \( y^*_i = \theta_{I_i} \) and \( x^*_j = \theta_{I_i} \) \( \forall j = \alpha_i, \ldots, b_i \);
    set \( x^*_j = \theta_j \) \( \forall j = b_i + 1, \ldots, \beta_i \);
end do

Figure 2: Algorithm to compute an optimal solution to LP1.

following observation:

**Observation 1.** An optimal solution to LP1 can be computed in \( O(n) \) time.

To reach the claimed complexity, it is enough to observe that, for each class \( i \in I \), the break item \( b_i \) can be computed in \( O(n_i) \) time. This can be done with a split argument similar to that proposed by [3] for finding the critical item in a KP, yielding an overall \( O(n) \) time complexity. As the resulting number of items to pack in the second step cannot be larger than \( n \), this gives a linear time algorithm.

We conclude this section showing a strengthened relaxation that exploits the fact that the optimal LP1 solution has at most one fractional \( y \) variable 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 original or cumulative) item that is selected at each iteration \( t \) of the Dantzig’s algorithm used to compute the LP1; 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 [16], we can derive the following upper bound for KPS

\[
UB = \max\{ UB_0, UB_1 \}
\]

where

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

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 \( UB \) 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, it is possible to 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 $y_i \geq 0 \ (i \in I)$.

**Observation 2.** There exists an optimal solution of RLP2, say $(x^*, y^*)$, such that

$$y_i^* = \frac{\sum_{j \in K_i} \frac{w_i}{C_i} x_j^*}{C_i} \quad \forall i \in I. \quad (21)$$

It is easy to see that, for a given class $i$, constraint (9) imposes the above lower bound for variable $y_i^*$, and that increasing $y_i^*$ with respect to this value produces a decrease of the solution value.

Based on Observation 2 one can reformulate RLP2 by substituting $y$ variables; this yields to the following model

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

where

$$\tilde{p}_j = p_j - \frac{f_{t_j}}{C_{t_j}} \tilde{w}_j \quad \text{and} \quad \tilde{w}_j = w_j + \frac{s_{t_j}}{C_{t_j}} \tilde{w}_j. \quad (22)$$

This model corresponds to the LP relaxation of a knapsack problem, which can be solved efficiently 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 (21).

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, our computational experiments showed that 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 $\xi_s$ variables ($i \in I, S \in \mathcal{S}_i$), 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, and refer the interested reader to [7] for further details on CG.
Model (14)–(16) and (18), 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:

$$WS\lambda + \pi_i \geq P_S$$  \hspace{1cm} i \in I, S \in \mathcal{S}_i, \quad (23)$$

where $\pi_i (i \in I)$ is the dual variable associated with the $i$-th constraint (16) and $\lambda$ is the dual variable associated with constraint (15). 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 $S^* \in \mathcal{S}_i$ for which the associated dual constraint (23), is violated, i.e., such that

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

(24)

where $\pi_i^* (i \in I)$ and $\lambda^*$ 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 $S^* \in \mathcal{S}_i$ that maximizes the left-hand-side of (24), 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 $\theta_j (j \in K_i)$, the problem reads as follows:

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

(25)

where $\theta_j = 1$ if and only if 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.

Since the pricing problems ask for the solution of a KP for each class, solving RMP (and, eventually, LP3) is weakly NP-hard. Computing an optimal solution for RMP at each CG iteration corresponds to solving the LP relaxation of the classical ILP formulation for the MCKP (see also section 2.3) for which a linear-time algorithm is known in the literature (see [13]). We use this algorithm to compute an optimal RMP primal solution and to identify the so-called critical item. Since in LP3, variables are associated to subset of items instead of items, we will denote the critical item as the critical subset. Finally, an optimal RMP dual solution can be characterized thanks to the following proposition:

**Proposition 3.** Let $\hat{S}$ be the critical subset in an optimal primal solution of RMP. Then, an optimal solution to the associated dual is the following:

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

**Proof.** It is easy to verify that $(\lambda^*, \pi^*)$ 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:

\[
\sum_{i \in I} \sum_{S \in \mathcal{A}_i} P_S \xi_S = \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathcal{A}_i} P_S \xi_S + P_{\hat{S}} \xi_{\hat{S}} - \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathcal{A}_i} P_S \xi_S + P_{\hat{S}} - P_{\hat{S}-1} \xi_{\hat{S}} + \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathcal{A}_i} P_S \xi_S + P_{\hat{S}} - 1 \xi_{\hat{S}} - 1 + \sum_{i \in I, i \neq \hat{i}} \sum_{S \in \mathcal{A}_i} P_S \xi_S + P_{\hat{S}} \xi_{\hat{S}}
\]

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

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

where the latter inequality derives from the definition of \( \pi_i^* \) 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 \((\pi_i^*, \lambda^*)\) 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(-and price) 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 macro-item. 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
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.

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 [18], which is the state-of-the-art for KP problems. 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, we observe that our scheme may require to branch on some \(y\) variable also in case all of them take an integer value in the current LP solution. The following example shows that this (apparently, 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 \(y_1 = y_2 = 1\), with items \(x_1 = x_3 = 1\) and \(x_2 = 0\); the optimal integer value is \(M\).

As already observed by [22], the algorithm by [1] does not allow branching on integer variables, and
may thus fail in finding the optimal solution 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 macro-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 be evaluated according to original profits and weights. Finally, for items that belong to the remaining classes one has to use profit and weights \( \tilde{p}_j \) and \( \tilde{w}_j \), respectively, see (22). 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 \( \tilde{p}_j \) and weight \( \tilde{w}_j \), 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 (16) in M3. To impose the condition \( y_i = 1 \), the constraint becomes:

\[
\sum_{s \in S_i} \xi_s = 1. \tag{26}
\]

On the other side, to impose the condition \( y_i = 0 \), the constraint becomes:

\[
\sum_{s \in S_i} \xi_s = 0. \tag{27}
\]

These modifications do not change the nature of the formulation nor the associated pricing problems PP. The effect of constraint (26) 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 give some details about our implementation of the algorithm. The new algorithm will be computationally tested in Section 7.

To reduce memory requirements, we implemented an algorithm that does not require the explicit storage of the two matrices, but uses two vectors of size \( C + 1 \) each. This is possible as, at each stage \( j \), only entries from column \( j - 1 \) are used to update the current stages. This space reduction may produce considerable improvements in terms of computing times too in practice, in particular for large-size problems. Our algorithm does not explicitly store the solution vector, but gives in output only the optimal solution value. This is coherent with what is done in the state-of-the-art DP for KPS, recently proposed in [21]; in particular, this is the DP algorithm used as benchmark in our computational experiments (see Section 7). As observed in [21], an optimal solution vector can be retrieved after the dynamic programming algorithm is completed, simply adapting the general recursive storage reduction principle from [20], 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

\[
s_i + \sum_{j \in K_i} w_j \leq C \quad \text{(28)}
\]

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 (28) 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 (28) is satisfied (see, [21]). Finally, this situation is always satisfied for the instances in our testbed that are taken from the literature (see Section 7.1).

Therefore, for the rest of this section we will assume that (28) is valid.

**Proposition 4.** Under assumption (28), \( U_1 \) dominates both \( U_A \) and \( U_B \).
Proof. Note that, for each class \( i \in I \), all items in \( K_i \) can be inserted in the knapsack. Thus, we have \( C^w_i = \sum_{j \in K_i} w_j \), 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 \( \pi_j \). To conclude the proof, we observe that, for the second instance in the proof of Proposition 1, upper bound \( U^1 \) is strictly smaller than both \( U^2_A \) and \( U^2_B \).

**Proposition 5.** Under assumption (28), 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 (25) for a given \( \lambda^* \) has the following optimal solution:

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

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 \([\alpha_i, j]\) is needed. 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 [8] and [23].

**Corollary 1.** Under assumption (28), \( U^1 = U^3 \).

**Proof.** This follows from Proposition 2 and by observing that, in case (28) is satisfied, constraints (19) are equivalent to their convex hull; thus, the convexified region has the integrality property. This implies that no bound improvement can be obtained by the convexification of constraints (4) and (19).

7. Computational Experiments

In this section we report an extensive computational analysis on the performances of our algorithms. All codes were implemented using C and were run on an Intel Xeon E3-1220 V2 running at 3.10 GHz. We first describe the instances taken from the literature (see Section 7.1) and some new randomly generated instances (see Section 7.2) and then present the outcome of our experiments.

7.1. Instances from the literature

To the best of our knowledge, some sets of test instances have been proposed in the literature for knapsack problems with setup. The 180 randomly generated instances proposed by [19] 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 [19]. 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 random 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 (e.g., [6] and [21]), and that all instances in this set satisfy condition (28).

Recently, four additional classes of instances have been introduced in [6]. The authors kindly provided us all these instances in a private communication. These instances, that were generated according to the scheme given in [22], have been denoted as Classes 1, 2, 3 and 5, whereas Class 6 includes 100 larger strongly correlated instances with up to 100000 items and 200 classes, where setup costs and capacities are obtained according to the scheme given in [22], have been denoted as Classes 1, 2, 3 and 5, whereas Class 6 includes 100 larger strongly correlated instances with up to 100000 items and 200 classes, where setup costs and capacities are obtained taking \( e_1 = e_2 \) uniformly ranging in the interval \([0.15, 0.25]\). For all classes proposed in [6], the capacity \( C \) is an integer value randomly generated in the range \([0.4(\sum_{j \in \mathcal{N}} w_j), 0.6(\sum_{j \in \mathcal{N}} w_j)]\). It turns out that all the instances in the new benchmark satisfy property (28).

7.2. New instances

To enlarge our benchmark we generated two additional classes of instances.

The first class (denoted as Class 6) is obtained from computationally challenging KP instances, in which the item weights and profits \((w_j \text{ and } p_j, j \in \mathcal{N})\) are generated using the generator proposed in [18] (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: \( w_j \text{ u.r. in } [1, R], p_j \text{ u.r. in } [1, R]. \)
2. Weakly correlated: \( w_j \text{ u.r. in } [1, R], p_j \text{ u.r. in } [\max\{1, w_j - R/10\}, w_j + R/10]. \)
3. Strongly correlated: \( w_j \text{ u.r. in } [1, R], p_j = w_j + R/10. \)
4. Inverse strongly correlated: \( p_j \text{ u.r. in } [1, R], w_j = p_j + R/10. \)
5. Almost strongly correlated: \( w_j \text{ u.r. in } [1, R], p_j \text{ u.r. in } [w_j + R/10 - R/500, w_j + R/10 + R/500]. \)
6. Subset-sum: \( w_j \text{ u.r. in } [1, R], p_j = w_j. \)
7. Even-odd subset-sum: $w_j$ even value u.r. in $[1, \bar{R}]$, $p_j = w_j$, $C$ odd.

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

where $\bar{R} = 1000$ 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 [6], 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 [6], 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 (28) discussed in Section 6.

The second class (denoted as Class 7) has been introduced to consider instances for which property (28) is violated. To the best of our knowledge, this is the first benchmark of instances with this characteristic. These instances are derived from the instances of Class 5, that are by far the hardest KPS problems from the existing literature. In particular, for each instance of Class 5, we define a twin instance in Class 7 as follows. We keep the same knapsack capacity and the same item set as in the original instance. Then, we include in the first class all the items of the first $\sigma_1$ classes in the original instance, where $\sigma_1 = \min\{k : \sum_{i=1}^{k} (s_i + \sum_{j \in K_i} w_j) > c\}$. The setup cost (resp. capacity) for the new class is the sum of the $\sigma_1$ setup costs (resp. capacities) of the original classes. In the same way, we define the second class by considering original classes $\sigma_1 + 1, \ldots, \sigma_2$, where $\sigma_2 = \min\{k : \sum_{i=\sigma_1+1}^{k} (s_i + \sum_{j \in K_i} w_j) > c\}$. In order to avoid instances with a too small number of classes (that could bias the comparison, as explained later), all the remaining classes $i = \sigma_2 + 1, \ldots, m$ are left unchanged.

All the instances considered in our computational analysis are available at the following address: http://or.dei.unibo.it/library/knapsack-problem-setup.

7.3. Solving the LP relaxation of the models

Our first set of experiments is aimed at evaluating the quality of the upper bound resulting from the LP relaxation of the ILP models given in Section 2. To evaluate the effect of condition (28), we consider both the instances proposed in [5] and the newly generated instances in Class 7.

Table 1 refers to instances of Class 4 and reports, for each model, the associated percentage gap, computed as $\% \text{gap} = 100 \times \frac{U - z^*}{z^*}$, where $U$ and $z^*$ denote the value of the relaxation and of the optimal solution, respectively. In this table we do not distinguish between LP1 and LP3, that produce the same upper bound (see, Corollary 1). Columns RLP2_A and RLP2_B correspond to the relaxations of models M2_A and M2_B, respectively (see Section 3.2). All figures report average values over the 50 instances having the same values for $m$.

These results confirm the theoretical dominance among the relaxations: LP1 and LP3 provide a very tight upper bound on the optimal value, while, on average, RLP2 (in both variants) yields a poor approximation. Finally, we note that while the combinatorial algorithms we used are extremely fast both from a theoretical and from a practical viewpoint, whereas solving these relaxations with
Table 1: LP relaxation of the models on instances of Class 4.

Table 2: LP relaxation of the models on instances of Class 7.

The results in Table 2 show that, for instances that do not satisfy condition (28), model M3 can produce a relaxation with an upper bound that is considerably tighter than the one provided by the relaxation of M1. Note however that the computation of this relaxation may be computationally more expensive to obtain than $U^1$, and this may lead to less efficient performances when large branch-and-bound trees are explored.

7.4. Combinatorial algorithms

In this section we evaluate the computational performances of exact approaches for KPS. In a set of preliminary computational experiments, we used IBM-ILOG Cplex 12.6 on models M1 and M2 and obtained results that are coherent with those reported in [5, 6]: the direct application of an ILP solver is typically unable to solve instances with more than 2500 items in a systematic way, even using state-of-the-art codes. For this reason, we do not report detailed results for this approach in what follows. As to the combinatorial algorithms, we consider B&B$_{LP1}$ and B&P$_{LP3}$, 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.

19
For this analysis, we used the instances of Class 4, and compared our algorithms with the exact algorithms given in [6] and in [21]. 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 [6] provided us with the implementation of their algorithm, which allows a better performance evaluation with our new exact algorithms. Since all our algorithms are sequential, in a first round of tests we ran DSS on our machine using 1 thread only and denote by DSS_{1TH} the resulting code (see the next section for a multi-thread performance analysis). We observe that the results for algorithm PS are taken from [21] and were obtained on an Intel i5 CPU running at 3.2 GHz with 16 GB of RAM.

<table>
<thead>
<tr>
<th>Instances</th>
<th>B&amp;B_{LP1}</th>
<th>B&amp;P_{LP3}</th>
<th>DP</th>
<th>DSS_{1TH}</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>5</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>1000</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td>2500</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.25</td>
</tr>
<tr>
<td>5000</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.90</td>
</tr>
<tr>
<td>10000</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>3.66</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>1000</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.06</td>
</tr>
<tr>
<td>2500</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.25</td>
</tr>
<tr>
<td>5000</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.93</td>
</tr>
<tr>
<td>10000</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
<td>3.70</td>
</tr>
<tr>
<td>20</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>1000</td>
<td>0.20</td>
<td>0.52</td>
<td>0.15</td>
<td>0.37</td>
<td>0.06</td>
</tr>
<tr>
<td>2500</td>
<td>3.19</td>
<td>5.98</td>
<td>1.42</td>
<td>2.74</td>
<td>0.25</td>
</tr>
<tr>
<td>5000</td>
<td>3.05</td>
<td>8.46</td>
<td>1.23</td>
<td>3.24</td>
<td>0.94</td>
</tr>
<tr>
<td>10000</td>
<td>7.07</td>
<td>14.63</td>
<td>2.61</td>
<td>5.18</td>
<td>3.76</td>
</tr>
<tr>
<td>30</td>
<td>0.00</td>
<td>0.02</td>
<td>0.01</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>1000</td>
<td>3.02</td>
<td>6.45</td>
<td>2.71</td>
<td>5.60</td>
<td>0.06</td>
</tr>
<tr>
<td>2500</td>
<td>18.95</td>
<td>70.47</td>
<td>10.00</td>
<td>37.77</td>
<td>0.25</td>
</tr>
<tr>
<td>5000</td>
<td>1.38</td>
<td>7.86</td>
<td>0.66</td>
<td>3.65</td>
<td>0.94</td>
</tr>
<tr>
<td>10000</td>
<td>11.90</td>
<td>78.98</td>
<td>4.87</td>
<td>30.83</td>
<td>3.76</td>
</tr>
</tbody>
</table>

Table 3: Sequential algorithms for the exact solution of KPS for instances of Class 4.

Table 3 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.

Results in Table 3 show that our enumerative algorithms are competitive with both DSS_{1TH} 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 (28) 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 disregards 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 [21]. 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.

7.5. Parallel algorithms

In this section we address the more challenging instances recently proposed in [6] as well as the new instances described in Section 7.2. 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 [6]), and denote by DSS4TH the resulting code. In our tests we used IBM-ILOG Cplex 12.6, that is a more recent release than the 12.5 one used in [6]. For this reason, computing times reported in Tables 4–6 for algorithm DSS4TH may be different (typically, smaller) than those reported in [6].

To have a fair comparison, we implemented a parallel variant of our algorithm (FMT in the following) that executes algorithms B&B_{LP1}, B&P_{LP3} and DP in parallel, and halts execution as soon as one of the 3 terminates. In particular, in our implementation we used OpenMP, an API that allows to add parallelism into existing source codes in a simple and flexible way. Furthermore, we report results for a parallel variant of algorithm DSS, in which this code is run in single-thread mode and one additional thread is used for running algorithm DP. We denote by DSS1TH+DP the resulting algorithm, which has been implemented using OpenMP as for FMT. The rationale of this algorithm is to allow a comparison with another parallel algorithm that exploits different strategies.

Table 4 reports the results on instances of Class 4, i.e., the classical KPS benchmark from the literature, proposed in [5] and already considered in Table 3. The table reports the same information as in Table 3, with an additional column (best) for algorithm FMT, in which a 3-field entry reports the number of instances for which B&B_{LP1}, B&P_{LP3} and DP is the best algorithm.

On these instances, algorithm FMT is typically faster than DSS4TH, with a considerable speedup for instances with $m \leq 10$. Observe that, in some cases, algorithm DSS4TH takes fully advantage of the parallelization and reaches a speedup that is even larger than the number of threads that are used. Indeed, modern ILP solvers adopt different strategies according to the number of available threads. For example, IBM-ILOG Cplex 12.6 includes cut generation procedures that are executed at the root node when multiple threads are available, and that are aimed at reducing the variability of the solver (see [9] for details). It is worth mentioning that FMT takes full advantage of its parallelism: for all instances with $m \leq 10$ the best algorithm is either B&B_{LP1} or B&P_{LP3}, whereas for large values of $m$ (in particular, for $n = 1000$ and $n = 2500$) DP is effective in reducing the computing time. Finally, we note that a similar behaviour does not occur for algorithm DSS1TH+DP: comparing
Table 4: Parallel algorithms on instances of Class 4.

<table>
<thead>
<tr>
<th>Instances</th>
<th>FMT</th>
<th></th>
<th>DSS_{4TH}</th>
<th></th>
<th>DSS_{1TH+DP}</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>avg</td>
<td>max</td>
<td>best</td>
<td>avg</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>0.00</td>
<td>0.00</td>
<td>10/0/0</td>
<td>0.09</td>
</tr>
<tr>
<td>1000</td>
<td>0.00</td>
<td>0.00</td>
<td>10/0/0</td>
<td>0.15</td>
<td>0.21</td>
</tr>
<tr>
<td>2500</td>
<td>0.00</td>
<td>0.01</td>
<td>10/0/0</td>
<td>0.35</td>
<td>0.40</td>
</tr>
<tr>
<td>5000</td>
<td>0.00</td>
<td>0.01</td>
<td>10/0/0</td>
<td>0.48</td>
<td>0.50</td>
</tr>
<tr>
<td>10000</td>
<td>0.01</td>
<td>0.01</td>
<td>10/0/0</td>
<td>0.85</td>
<td>0.98</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>0.00</td>
<td>0.01</td>
<td>10/0/0</td>
<td>0.14</td>
</tr>
<tr>
<td>1000</td>
<td>0.01</td>
<td>0.01</td>
<td>0/10/0</td>
<td>0.16</td>
<td>0.20</td>
</tr>
<tr>
<td>2500</td>
<td>0.01</td>
<td>0.02</td>
<td>0/10/0</td>
<td>0.35</td>
<td>0.42</td>
</tr>
<tr>
<td>5000</td>
<td>0.01</td>
<td>0.02</td>
<td>5/5/0</td>
<td>0.54</td>
<td>0.58</td>
</tr>
<tr>
<td>10000</td>
<td>0.02</td>
<td>0.03</td>
<td>3/7/0</td>
<td>0.90</td>
<td>0.96</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>0.00</td>
<td>0.00</td>
<td>10/0/0</td>
<td>0.14</td>
</tr>
<tr>
<td>1000</td>
<td>0.06</td>
<td>0.06</td>
<td>0/2/8</td>
<td>0.26</td>
<td>0.43</td>
</tr>
<tr>
<td>2500</td>
<td>0.23</td>
<td>0.25</td>
<td>0/1/9</td>
<td>0.44</td>
<td>0.74</td>
</tr>
<tr>
<td>5000</td>
<td>0.68</td>
<td>0.98</td>
<td>0/5/5</td>
<td>0.65</td>
<td>0.74</td>
</tr>
<tr>
<td>10000</td>
<td>2.41</td>
<td>3.88</td>
<td>0/7/3</td>
<td>1.01</td>
<td>1.12</td>
</tr>
<tr>
<td>30</td>
<td>500</td>
<td>0.00</td>
<td>0.02</td>
<td>9/0/1</td>
<td>0.40</td>
</tr>
<tr>
<td>1000</td>
<td>0.06</td>
<td>0.06</td>
<td>0/0/10</td>
<td>0.53</td>
<td>1.37</td>
</tr>
<tr>
<td>2500</td>
<td>0.23</td>
<td>0.26</td>
<td>0/1/9</td>
<td>0.49</td>
<td>0.59</td>
</tr>
<tr>
<td>5000</td>
<td>0.36</td>
<td>0.98</td>
<td>0/8/2</td>
<td>0.79</td>
<td>1.05</td>
</tr>
<tr>
<td>10000</td>
<td>1.65</td>
<td>3.89</td>
<td>0/7/3</td>
<td>1.59</td>
<td>2.32</td>
</tr>
</tbody>
</table>

its results with those of algorithm DSS_{1TH} of Table 3, it can be noticed that DP yields a reduction of the computing times only for instances with small values of $n$, whereas for the hardest instances a 2x slowdown is observed. This computational behaviour can be explained by the fact that, for large instances, both IBM-ILOG Cplex 12.6 and DP may require a large amount of memory, which deteriorates the performances. Since a similar behaviour is experienced for all the instances in our benchmark, we refrain from reporting any further result for algorithm DSS_{1TH+DP} hereafter.

Tables 5 and 6 report the results for all the classes of instances proposed in [6]. Each line of these tables gives the average and the maximum computing time over ten instances with similar characteristics.

In Table 5 we compare the performances of FMT and DSS_{4TH} 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 7.1. These results show that our algorithm FMT is competitive with algorithm DSS_{4TH}: for instances of Class 1, FMT outperforms DSS_{4TH}, 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_{4TH} in 431 cases.

Results on instances of Class 5 are given in Table 6. On this benchmark, algorithm DSS_{4TH} 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\textsubscript{4TH} are highly influenced by the number of items. For this reason, our approach turns out to be much faster than DSS\textsubscript{4TH} 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\textsubscript{4TH} 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.

We now present the results for the new instances introduced in Section 7.2. Since these new instances are intended to represent challenging benchmarks, for all these experiments we set a time limit of 1200 CPU seconds per instance.

Each row in Table 7 shows 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\textsubscript{4TH} we report the number of instances solved to proven optimality within the time limit, and the average and maximum computing times. Average computing times refer to instances solved to proven optimality only. T.L. means that at least one instance in the associated subset hits the given time limit.

These results confirm that for hard instances with a small number of classes algorithm FMT outperforms DSS\textsubscript{4TH}. Indeed, the former is able to solve each of the 240 instances in Class 6 in at most half a second, whereas algorithm DSS\textsubscript{4TH} fails in solving 6 problems within the 1200-seconds time limit. As to the average computing times, FMT is always faster than DSS\textsubscript{4TH}, often by some

<table>
<thead>
<tr>
<th>Instances</th>
<th>FMT</th>
<th>DSS\textsubscript{4TH}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>50 [40-60]</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>50 [90-110]</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>100 [40-60]</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>100 [90-110]</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 5: Parallel algorithms on instances of Classes 1, 2 and 3.
<table>
<thead>
<tr>
<th>Instances</th>
<th>FMT</th>
<th>DSS$_{4TH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>avg</td>
</tr>
<tr>
<td>5</td>
<td>20000</td>
<td>0.01</td>
</tr>
<tr>
<td>10000</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>0.11</td>
<td>10/0/0</td>
</tr>
<tr>
<td>10</td>
<td>20000</td>
<td>0.03</td>
</tr>
<tr>
<td>50000</td>
<td>0.07</td>
<td>0.09</td>
</tr>
<tr>
<td>100000</td>
<td>0.12</td>
<td>0.16</td>
</tr>
<tr>
<td>20</td>
<td>20000</td>
<td>4.29</td>
</tr>
<tr>
<td>50000</td>
<td>9.43</td>
<td>22.85</td>
</tr>
<tr>
<td>100000</td>
<td>17.58</td>
<td>44.03</td>
</tr>
<tr>
<td>30</td>
<td>20000</td>
<td>4.10</td>
</tr>
<tr>
<td>50000</td>
<td>23.10</td>
<td>106.92</td>
</tr>
<tr>
<td>100000</td>
<td>120.25</td>
<td>429.79</td>
</tr>
<tr>
<td>50</td>
<td>20000</td>
<td>7.46</td>
</tr>
<tr>
<td>50000</td>
<td>53.22</td>
<td>107.70</td>
</tr>
<tr>
<td>100000</td>
<td>85.02</td>
<td>432.17</td>
</tr>
<tr>
<td>100</td>
<td>20000</td>
<td>9.32</td>
</tr>
<tr>
<td>50000</td>
<td>47.82</td>
<td>108.02</td>
</tr>
<tr>
<td>100000</td>
<td>161.53</td>
<td>432.80</td>
</tr>
<tr>
<td>200</td>
<td>20000</td>
<td>12.50</td>
</tr>
<tr>
<td>50000</td>
<td>58.44</td>
<td>108.75</td>
</tr>
<tr>
<td>100000</td>
<td>224.17</td>
<td>435.15</td>
</tr>
</tbody>
</table>

Table 6: Parallel algorithms on instances of Class 5.

<table>
<thead>
<tr>
<th>Instances</th>
<th>FMT</th>
<th>DSS$_{4TH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>#opt</td>
</tr>
<tr>
<td>5</td>
<td>5000</td>
<td>24</td>
</tr>
<tr>
<td>10000</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>20000</td>
<td>24</td>
<td>0.02</td>
</tr>
<tr>
<td>100000</td>
<td>24</td>
<td>0.03</td>
</tr>
<tr>
<td>10</td>
<td>5000</td>
<td>24</td>
</tr>
<tr>
<td>10000</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>20000</td>
<td>24</td>
<td>0.01</td>
</tr>
<tr>
<td>100000</td>
<td>24</td>
<td>0.04</td>
</tr>
<tr>
<td>200000</td>
<td>24</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 7: Parallel algorithms on instances of Class 6.

orders of magnitude. Indeed, though we compute average values only considering instances solved
to proven optimality, FMT has an average computing time of 0.02 seconds, while DSS$_{4TH}$ requires
about 50 seconds on average. Finally, we observe that, for these instances, DP is never the best algorithm; this may be expected as problems in Class 6 have a large number of items but a relatively small number of classes, and thus can be solved efficiently using combinatorial algorithms B&B\textsubscript{LP1} and B&P\textsubscript{LP3}.

Table 8 reports the results of our experiments on the instances of Class 7. Since these problems are derived from those of Class 5, the instances are grouped as in Table 6, i.e., according to the number of items $n$ and to the number of classes $m$ in the original instances. For each group of 10 instances we also give the average number $m^*$ of classes in the new instances; this number is in the range from 2 to 35.

<table>
<thead>
<tr>
<th>Instances</th>
<th>FMT</th>
<th>DSS\textsubscript{4TH}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>$m$</td>
<td>$n$</td>
<td>$m^*$</td>
</tr>
<tr>
<td>5</td>
<td>20000</td>
<td>2.0</td>
</tr>
<tr>
<td>50000</td>
<td>2.0</td>
<td>0.01</td>
</tr>
<tr>
<td>100000</td>
<td>2.0</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>20000</td>
<td>2.0</td>
</tr>
<tr>
<td>50000</td>
<td>2.2</td>
<td>0.01</td>
</tr>
<tr>
<td>100000</td>
<td>2.3</td>
<td>0.02</td>
</tr>
<tr>
<td>20</td>
<td>20000</td>
<td>4.0</td>
</tr>
<tr>
<td>50000</td>
<td>4.0</td>
<td>0.01</td>
</tr>
<tr>
<td>100000</td>
<td>4.2</td>
<td>0.02</td>
</tr>
<tr>
<td>30</td>
<td>20000</td>
<td>5.8</td>
</tr>
<tr>
<td>50000</td>
<td>5.8</td>
<td>0.01</td>
</tr>
<tr>
<td>100000</td>
<td>6.0</td>
<td>0.04</td>
</tr>
<tr>
<td>50</td>
<td>20000</td>
<td>9.0</td>
</tr>
<tr>
<td>50000</td>
<td>9.6</td>
<td>0.04</td>
</tr>
<tr>
<td>100000</td>
<td>9.1</td>
<td>0.08</td>
</tr>
<tr>
<td>100</td>
<td>20000</td>
<td>19.0</td>
</tr>
<tr>
<td>50000</td>
<td>17.6</td>
<td>0.48</td>
</tr>
<tr>
<td>100000</td>
<td>17.3</td>
<td>0.97</td>
</tr>
<tr>
<td>200</td>
<td>20000</td>
<td>35.5</td>
</tr>
<tr>
<td>50000</td>
<td>35.1</td>
<td>67.16</td>
</tr>
<tr>
<td>100000</td>
<td>34.8</td>
<td>196.36</td>
</tr>
</tbody>
</table>

Table 8: Parallel algorithms on instances of Class 7.

The results in Table 8 show that FMT is typically faster than DSS\textsubscript{4TH} also for instances for which condition (28) does not hold. These instances have the same computational difficulty as those in Class 5 for algorithm FMT, and in this case too our algorithm largely outperforms DSS\textsubscript{4TH} (in some cases, up to 4 orders of magnitude) for problems with a small number of classes. Finally observe that, contrary to the instances of Class 5, algorithm B&P\textsubscript{LP3} is never the best algorithm among those used within FMT.
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 [19], 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.

Acknowledgments

The authors thank Federico Della Croce, Fabio Salassa and Rosario Scatamacchia for making the source code for their algorithms available. Thanks are also due to three anonymous referees for helpful comments. The research of Michele Monaci has been supported by Air Force Office of Scientific Research, grant FA9550-17-1-0067.

References


