Version of Record: https://www.sciencedirect.com/science/article/pii/S0377221718309111
Manuscript_e560bb1216977f1fcf6cc198c1256bba

Mathematical models and decomposition methods for the multiple knapsack problem

Mauro Dell'Amico[(1)], Maxence Delorme[(2)], Manuel Iori[(1)], Silvano Martello[(3)]

(1) DISMI, University of Modena and Reggio Emilia
(2) School of Mathematics, University of Edinburgh
(3) DEI "Guglielmo Marconi", University of Bologna

Corresponding author maxence.delorme@ed.ac.uk, phone +44 0131 650 5870

Abstract

We consider the multiple knapsack problem, that calls for the optimal assignment of a set of items, each having a profit and a weight, to a set of knapsacks, each having a maximum capacity. The problem has relevant managerial implications and is known to be very difficult to solve in practice for instances of realistic size. We review the main results from the literature, including a classical mathematical model and a number of improvement techniques. We then present two new pseudo-polynomial formulations, together with specifically tailored decomposition algorithms to tackle the practical difficulty of the problem. Extensive computational experiments show the effectiveness of the proposed approaches.

Keywords: Combinatorial optimization, Multiple knapsack problem, Exact algorithms, Pseudo-polynomial formulations, Decomposition methods.

1 Introduction

Given a set of m containers (*knapsacks*) with *capacity* c_i $(i = 1, \dots, m)$ and a set of n objects (*items*) with *profit* p_j and *weight* w_j $(j = 1, \dots, n)$, we consider the *Multiple Knapsack Problem* (MKP): select m disjoint subsets of items (one per knapsack) such that the total weight of the items in the knapsack does not exceed its capacity, and the overall profit of the selected items is a maximum.

The MKP has a number of important special cases in the *Cutting and Packing* area. The most famous one, arising when $m = 1$, is the (single) *Knapsack Problem*, for which a huge literature exists (see, e.g., the specific chapters in Martello and Toth [42] and in Kellerer, Pferschy, and Pisinger [33]). When $p_j = w_j$ for $j = 1, \dots, n$, we have the *Multiple Subset-Sum Problem* (MSSP), that has been studied both in its natural version and in the special case where all capacities are equal (see Caprara, Kellerer, and Pferschy [5, 6]). When, in addition, there is a unique knapsack, the MSSP is known as the (single) *Subset-Sum Problem* (see again the specific chapters in [42] and [33]). Finally, an MKP in which all items have the same profit and all knapsacks have the same capacity is known as the *Maximum Cardinality Bin Packing Problem* (see, e.g., Labbé, Laporte, and Martello [36]).

A relevant generalization of the MKP, the *Generalized Assignment Problem*, arises when the profit of each item depends on the knapsack it is assigned to, i.e., the profits are p_{ij} $(i = 1, \dots, m; j = 1, \dots, n)$.

The mentioned single-container problems (knapsack and subset-sum) can be shown to be weakly $\mathcal{NP}$-hard by transformation from the partition problem (see Karp [31]), while all the others are

known to be strongly $\mathcal{NP}$-hard by transformation from the 3-partition problem (see Garey and Johnson [26]).

According to the classification scheme proposed by Dyckhoff [17], the MKP can be indicated as `1/B/D/-` (or as `1/B/I/-`, when all knapsacks have the same capacity). The problem is called the *1-dimensional Multiple Heterogeneous Knapsack Problem* (MHKP) in the typology proposed by Wäscher, Haußner, and Schumann [50].

The MKP has important real world managerial applications. In their seminal paper [18], Eilon and Christofides mentioned applications in vehicle and container loading. Ferreira, Martins, and Weismantel [20] described real world problems in the design of processors for mainframe computers, in the layout of electronic circuits, and in sugar cane alcohol production in Brazil that could be solved through a generalization of the MKP. Kalagnanam, Davenport, and Lee [30] used the MKP within a complex approach for clearing continuous call double-sided auctions in the case of indivisible demands. Recently, Simon, Apte, and Regnier [48] modeled, through MKPs, problems of maintaining operational capability without external support (typically arising in humanitarian assistance and disaster relief as well as in military operations). Other applications are mentioned in [33].

In the next section we review the main results from the MKP literature. Section 3 contains a basic mathematical model and the description of a number of improvement techniques. In Section 4 we present pseudo-polynomial formulations for the MKP, while decomposition approaches are introduced in Section 5. Our best exact method, a hybrid combination of several techniques, is given in Section 6. The results of extensive computational experiments on the various approaches are reported and commented in Section 7.

2 Literature review

In this section we briefly describe the main contributions to the solution of the MKP. The reader is referred to the book chapters below for more exhaustive presentations.

Book chapters

As mentioned in the previous section, two chapters in the books by Martello and Toth [42] and Kellerer, Pferschy, and Pisinger [33] deal with the MKP. They review the existing literature at the time of publication, and the main solution approaches: upper bounds, exact methods, heuristics, approximation algorithms, and polynomial-time approximation schemes.

Heuristics

Fisk and Hung [21] were probably the first to present a heuristic algorithm for the MKP. It consists of a non-polynomial method that: (i) exactly solves the *surrogate relaxation* obtained by replacing the m knapsacks with a single knapsack with capacity equal to the sum of the knapsack capacities; (ii) obtains a feasible solution by trying to insert the selected items into the knapsacks through greedy insertions and local exchanges.

Martello and Toth [41] proposed various polynomial-time heuristics based on greedy algorithms and local search procedures helped by the rearrangement of the greedy solutions. The resulting overall algorithm is known as MTHM, and the corresponding Fortran computer code is available at `http://www.or.deis.unibo.it/knapsack.html`. The algorithms were tested on instances with up to 1000 items and 100 knapsacks, and later on, in [42], on larger instances with up to 10 000 items and 40 knapsacks.

Lalami, Elkihel, El Baz, and Boyer [37] proposed a heuristic that recursively solves the so-called *core* (see Balas and Zemel [1]) of different single knapsack problems through dynamic programming. They compared it with MTHM on randomly generated instances with up to 100 000 items and 100 knapsacks, obtaining better gaps and smaller CPU times. Worth is noting that the reported gaps are always below 0.2% (and usually below 0.01%), both for MTHM and the proposed algorithm.

Fukunaga [22] and Fukunaga and Tazoe [25] presented various genetic approaches and tested them on instances with up to 300 items and 100 knapsacks. They obtained consistent improvement in terms of solution quality with respect to MTHM, although at the expenses of much larger execution times.

Laalaoui [34] experimented two swap heuristics to improve on the solutions produced by MTHM. The approach was later extended by Laalaoui and M'Hallah [35], who proposed a variable neighborhood search that makes use of a linked list data structure and a dynamic threshold acceptance criterion. They computationally tested their algorithm on instances with up to 4800 items and 2400 knapsacks, obtaining state of the art results improving both on MTHM and the genetic algorithms by Fukunaga [22].

It is known (see [42], Section 1.3) that the MKP cannot have a *fully polynomial-time approximation scheme*. *Polynomial-time approximation schemes* were developed by Chekuri and Khanna [8] for the MKP and by Caprara, Kellerer, and Pferschy [6] for the MSSP. For the special MSSP case in which all knapsacks have the same capacity, Caprara, Kellerer, and Pferschy [7] proposed a polynomial-time $\frac{3}{4}$-approximation algorithm.

Enumerative algorithms

The first *branch-and-bound* algorithms for the MKP were proposed by Ingargiola and Korsh [29], Hung and Fisk [28], and Martello and Toth [39]. Later on, Martello and Toth [40] proposed a special enumerative algorithm (*bound and bound* method), whose Fortran implementation (known as MTM, and available at `http://www.or.deis.unibo.it/knapsack.html`) turned out to be computationally much faster than the previous approaches. These algorithms include upper bound computations, obtained through surrogate and Lagrangian relaxations of the capacity constraints, that were studied in [40] and [42], respectively.

Later on, Pisinger [44] derived from MTM a more efficient exact procedure, MULKNAP, capable of solving to optimality large-size instances with up to 100 000 items and 10 knapsacks. More recently, Fukunaga and Korf [23, 24] improved the performance of the MULKNAP bound and bound algorithm by integrating techniques from constraint programming and artificial intelligence in a solver that appears to be particularly effective on instances characterized by high $\frac{n}{m}$ ratios (but has more difficulties for instances with smaller ratios). Computational tests with a heterogeneous aggregation for Dantzig-Wolfe reformulation were reported by Bergner and Dahms [2].

3 Mathematical model and preprocessing techniques

A classical, intuitive, model for the MKP can be defined by introducing binary variables

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is packed into knapsack } i; \\ 0 & \text{otherwise,} \end{cases}$$

as follows (see [42]):

$$\max \quad z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j \; x_{ij} \tag{1}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j \, x_{ij} \leq c_i \qquad i = 1, \ldots, m, \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad j = 1, \ldots, n, \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, m \quad j = 1, \ldots, n. \tag{4}$$

Objective function (1) maximizes the profit of the packed items. Constraints (2) impose that the capacity of each knapsack is respected, while constraints (3) ensure that each item is packed in at most one knapsack. Without loss of generality we assume that each knapsack can contain at least one item and that each item can be contained in at least one knapsack (i.e., that $\min_j\{w_j\} \leq \min_i\{c_i\}$ and $\max_j\{w_j\} \leq \max_i\{c_i\}$). An equivalent model can be obtained by: (i) defining binary variables t_j $(j = 1, \ldots, n)$ taking the value 1 if item j is selected and the value 0 otherwise; (ii) adding constraints $t_j = \sum_{i=1}^{m} x_{ij}$ $(j = 1, \ldots, n)$, and (iii) replacing variables x with variables t in the objective function. The new model may exhibit a different behavior with respect to the basic one, as will be discussed in Section 7.

The computational performance of algorithms based on model (1)-(4), as well as on other models discussed in the following, can be increased through a number of improvement techniques. We briefly describe in the following the main techniques from the literature that were adopted or adapted to the proposed algorithms.

Instance reduction

It is quite common in cutting and packing problems to preliminarily use special techniques in order to decide the optimal value of a subset of variables and hence reducing the instance size. We describe here an intuitive adaptation of instance reduction to the MKP, that has been recently used by Martello and Monaci [38].

Let I be any subset of knapsacks and let J be the set of all items that can be packed in a knapsack of I, i.e., $J := \{j : w_j \leq \max_{i \in I}\{c_i\}, 1 \leq j \leq n\}$: if there exists a feasible packing of the items of J into the knapsacks of I, then such packing can be fixed and sets I and J can be removed from the instance.

In order to efficiently implement this property, it is convenient to sort the knapsacks by non-decreasing capacity, start with $I = \{1\}$, and iteratively add the next (smaller) knapsack to I. At each iteration, we include the appropriate additional items into J, and check if $\sum_{j \in J} w_j \leq \sum_{i \in I} c_i$.

If the answer is yes, we invoke a *Constraint Programming* (CP) approach that is based on the bin packing constraint introduced by Shaw [47]. Such constraint is formally written as $\mathcal{P}(l, b, w)$, where: (i) l is a vector of m constrained variables representing the load of each bin; (ii) b is a vector of n constrained variables indicating, for each item, the index of the bin in which it will be placed; and (iii) w is a vector of n non-negative integers $(w_1, w_2, \ldots, w_n)$ representing the weight of each item to be packed. In our tests, we used the implementation provided by Cplex in the `IloPack` constraint. This CP method will also be used in Section 5.1 to implement a decomposition algorithm.

Capacity lifting

Lifting techniques are also widely used in the cutting and packing area. We adopt the capacity lifting method proposed by Pisinger [44]. For each knapsack i, we determine, through dynamic programming, the maximum total weight c'_i that can be obtained by packing any subset of items into i: if $c'_i < c_i$ then the capacity of the knapsack is set to c'_i.

Item dominance

A simple item dominance criterion was included by Ingargiola and Korsh [29] in their branch-and-bound algorithm for the MKP. In a knapsack problem, given two items j and k, if $w_j \leq w_k$ and $p_j \geq p_k$, then we say that j *dominates* k (for any feasible solution that includes k but excludes j, there is a better solution that excludes k and includes j). Hence, when a branch decision excludes an item from the solution, all items dominated by it can also be excluded. We implemented this idea by preliminarily sorting the items according to non-increasing weight, breaking ties by non-decreasing profit. For each item k, items $j := k+1, \ldots, n$ are scanned and, if $p_j \geq p_k$ then the pair (j, k) is added to a dominance list $\mathcal{D}$. Model (1)-(4) can then be enforced by adding the set of constraints

$$\sum_{i=1}^{m} x_{ij} \geq \sum_{i=1}^{m} x_{ik} \quad \text{for each} \quad (j,k) \in \mathcal{D}. \tag{5}$$

The three improvement procedures described above were included in all algorithmic implementations introduced in the next sections, and used in all our tests. We mention however that, for very large knapsack capacities, it could be advisable to avoid using capacity lifting, as it could be time-consuming with little effect.

4 Pseudo-polynomial formulations

Model (1)-(4) has a polynomial number of variables, nm. In this section, we present two pseudo-polynomial models, whose number of variables also depends on the largest knapsack capacity $c = \max_{i=1,\ldots,m}\{c_i\}$.

The use of pseudo-polynomial models to solve cutting and packing problems dates back at least to the 1970s, when Rao [46] and, independently, Wolsey [51] used them to solve the classical *Cutting Stock Problem* (CSP). Since then, a number of effective pseudo-polynomial models has been produced, to solve not only the CSP but also other relevant variants. Among these, we cite methods *One-cut* by Dyckhoff [16] and Rao [46], *Arc-flow* by Valério de Carvalho [49], *DP-flow* by Cambazard and O'Sullivan [4], *VPsolver* by Brandão and Pedroso [3], *Meet-in-the-Middle* by Côté and Iori [12], and *Reflect* by Delorme and Iori [13]. We refer the interested reader to Delorme, Iori and Martello [14] for a recent survey on exact algorithms and mathematical models for the CSP. Very recently, Martinovic et al. [43] and Delorme and Iori [13] independently proved that One-cut, Arc-flow and the pattern-based model by Gilmore and Gomory [27] are equivalent one another, closing an open question in the field of cutting and packing.

In the following, we present two new formulations that can effectively tackle the MKP and are based, respectively, on Arc-flow and Reflect.

4.1 Arc-flow model

The general idea of Arc-flow is to consider the filling of a knapsack as a path in a graph where nodes are partial knapsack fillings and arcs are items. The aim is to find a set of paths, one for each knapsack, that contains a subset of items having maximum profit. Formally, let $c = \max_{i=1,\ldots,m} c_i$ and $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a multi-graph having node set $\mathcal{N} = \{0, 1, \ldots, c, c+1\}$. To correctly represent a knapsack packing, each path must start at the source node $s = 0$ and terminate at the sink node $t = c + 1$. Arcs in $\mathcal{A}$ are of the form (d, e, j, i), where d and e are two nodes representing (partial) knapsack fillings, j either is an item index or takes the value 0, and i either is a knapsack index or takes the value 0. We consider three subsets of arcs, partitioning $\mathcal{A}$ into $\mathcal{A}_s \cup \mathcal{A}_k \cup \mathcal{A}_l$ as follows:

Algorithm 1 `Create_arcflow_multigraph_MKP`

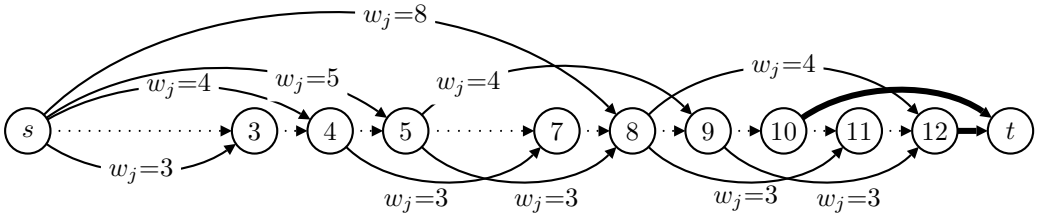
1: **Input:** m: number of knapsacks; c_i: knapsack capacities; n: number of items; w_j: item weights
2: $\mathcal{N} \leftarrow \emptyset$; $\mathcal{A}_s \leftarrow \emptyset$; $\mathcal{A}_k \leftarrow \emptyset$; $\mathcal{A}_l \leftarrow \emptyset$; $s \leftarrow 0$; $c \leftarrow \max_{i=1,\ldots,m}\{c_i\}$; $t \leftarrow c+1$
3: $T[l] \leftarrow 0$ **for** $l = 1, \ldots, c$ ▷ an array that keeps track of the possible tails
4: $T[s] \leftarrow 1$; $\mathcal{N} \leftarrow \mathcal{N} \cup \{s\}$ ▷ node s is the first tail
5: **for** $j = 1$ **to** n **do** ▷ item arcs
6: **for** $l = c - w_j$ **down to** 0 **do**
7: **if** $T[l] = 1$ **then** ▷ if l is a possible tail
8: $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup \{(l, l+w_j, j, 0)\}$
9: $T[l+w_j] \leftarrow 1$; $\mathcal{N} \leftarrow \mathcal{N} \cup \{(l+w_j)\}$
10: **end if**
11: **end for**
12: **end for**
13: **for** $i = 1$ **to** m **do** ▷ knapsack arcs
14: $\mathcal{A}_k \leftarrow \mathcal{A}_k \cup \{(c_i, t, 0, i)\}$
15: $\mathcal{N} \leftarrow \mathcal{N} \cup \{c_i\}$
16: **end for**
17: **for** $l \in \mathcal{N}$ **do** $\mathcal{A}_l \leftarrow \mathcal{A}_l \cup \{(l, l', 0, 0)$: $l' = \min(e \in \mathcal{N} : c \geq e > l)\}$ ▷ loss arcs
18: $\mathcal{A} \leftarrow \mathcal{A}_s \cup \mathcal{A}_k \cup \mathcal{A}_l$
19: **return** $\mathcal{N}, \mathcal{A}$

- $\mathcal{A}_s = \{(d, d+w_j, j, 0)$ with $j \in \{1, \ldots, n\}\}$ is the set of *item arcs* that are used to represent the assignment of item j to a knapsack having current filling $d \in \{0, \ldots, c - w_j\}$;
- $\mathcal{A}_k = \{(c_i, t, 0, i)$ with $i \in \{1, \ldots, m\}\}$ is the set of *knapsack arcs* that model the use of knapsack i;
- $\mathcal{A}_l = \{(d, e, 0, 0)\}$ is the set of *loss arcs* that represent unused space from d to e, with $d < e$.

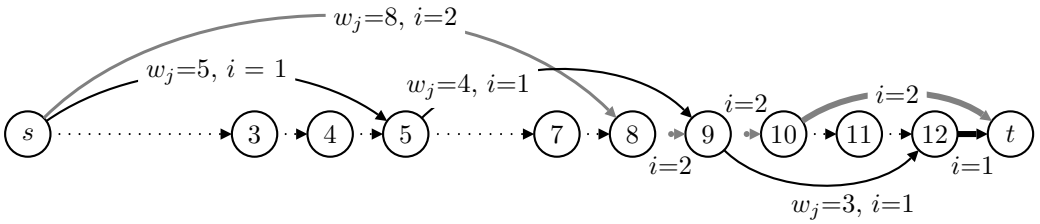
We provide in Algorithm 1 the procedure used to construct sets $\mathcal{A}$ and $\mathcal{N}$. The procedure is an adaptation to the MKP of the classical arc generation algorithm used for the CSP (see, e.g., [12], [13], and [49]). To keep the resulting number of arcs low, it is computationally convenient to preliminarily sort the items by non-increasing weight. For each item j the algorithm generates an item arc only if it starts from an 'active' node . A node d is *active* if it represents a complete filling of d units, that could be obtained with the items preceding j (see, e.g., Valério de Carvalho [49] for details). Algorithm 1 uses an array T to keep track of the possible arc tails. Initially, T contains a single possible tail s (Step 4). Then (Step 5) for each item j the algorithm scans all possible tails l that could lead to a valid arc (Step 6) and creates an item arc from l to $l + w_j$ (Step 8). Node $l + w_j$ is then added to the possible tails and to the set of nodes (Step 9). Once all items arcs have been built, the algorithm creates the knapsack arcs (Step 14), adds nodes c_i to $\mathcal{N}$ (Step 15), and terminates by adding loss arcs connecting all pairs of consecutive nodes in $\mathcal{N}$ (Step 17).

Figure 1-(a) depicts the graph created by Algorithm 1 for a small instance composed by 2 knapsacks of capacity 12 and 10 and 4 items of weights 8, 5, 4, and 3. The purpose of the example is to help understanding how the Arc-flow graph is created, so, to keep it as simple as possible, we disregard the item profits and consider a case in which all items are inserted into the knapsacks. Algorithm 1 starts by creating item arcs (0, 8, 1, 0) and (0, 5, 2, 0) for the two first items, item arcs (0, 4, 3, 0), (5, 9, 3, 0), and (8, 12, 3, 0) for the third item, and item arcs (0, 3, 4, 0), (4, 7, 4, 0), (5, 8, 4, 0), (8, 11, 4, 0), and (9, 12, 4, 0) for the fourth item. It then creates knapsack arcs

$(12, t, 0, 1)$ for knapsack 1 and $(10, t, 0, 2)$ for knapsack 2 (depicted in thick lines). It terminates by creating loss arcs (depicted as horizontal dotted lines) between any pair of consecutive nodes. Figure 1-(b) depicts an optimal solution in which items 2, 3, and 4 are packed together in the knapsack of capacity 12 and item 1 in the knapsack of capacity 10.



(a) Arc-flow graph



(b) Arc-flow solution

Figure 1: Arc-flow graph and optimal solution for an instance with 2 knapsacks and 4 items. Item arcs are depicted in solid lines, knapsack arcs in thick lines, and loss arcs in dotted lines.

To formally describe our model, let $\delta^+(e)$ (respectively, $\delta^-(e)$) be the subset of arcs emanating from (respectively, entering) node e. By introducing an integer variable x_{deji}, giving the number of times arc (d, e, j, i) is chosen, and a binary variable t_j, taking the value 1 iff item j is selected, the MKP can be modeled as:

$$\max \quad \sum_{j=1}^{n} p_j t_j \tag{6}$$

$$\text{s.t.} \quad \sum_{(e,f,j,i)\in\delta^+(e)} x_{efji} - \sum_{(d,e,j,i)\in\delta^-(e)} x_{deji} = \begin{cases} m & \text{if } e = s \\ -m & \text{if } e = t \\ 0 & \text{otherwise} \end{cases} \quad e \in \mathcal{N}, \tag{7}$$

$$\sum_{(d,d+w_j,j,0)\in\mathcal{A}_s} x_{d,d+w_j,j,0} \geq t_j \qquad j = 1, \ldots, n, \tag{8}$$

$$x_{deji} \in \mathbb{N} \qquad (d, e, j, i) \in \mathcal{A}_l, \tag{9}$$

$$x_{deji} \in \{0, 1\} \qquad (d, e, j, i) \in \mathcal{A}_s \cup \mathcal{A}_k, \tag{10}$$

$$t_j \in \{0, 1\} \qquad j = 1, \ldots, n. \tag{11}$$

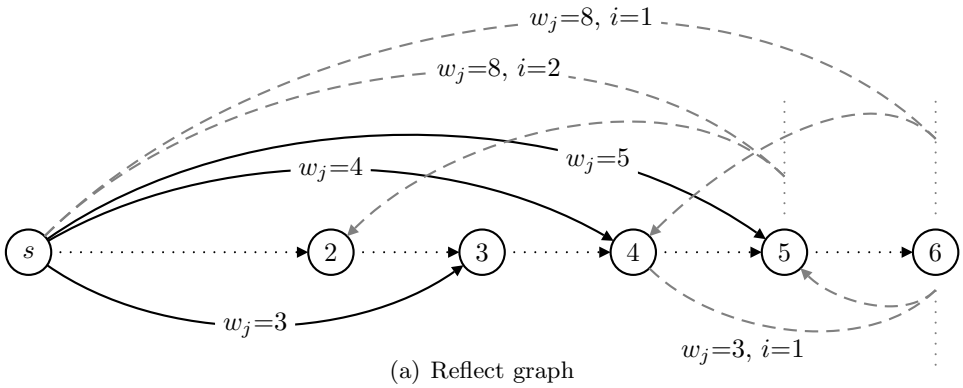
Objective function (6) maximizes the profit of the packed items. Constraints (7) ensure the flow conservation. Constraints (8) impose that if an item is selected in the solution, then an arc corresponding to such item is also selected. Constraints (10) limit to 1 the use of each knapsack. Note that variables t_j are not strictly necessary, as t_j could be replaced by $\sum_{(d,d+w_j,j,0)\in\mathcal{A}_s} x_{d,d+w_j,j,0}$ in (6), and (8) could be replaced by $\sum_{(d,d+w_j,j,0)\in\mathcal{A}_s} x_{d,d+w_j,j,0} \leq 1$. However, their use makes the

model easier to read and also computationally more efficient, as, according to our tests, they help MIP solvers in obtaining a faster convergence (as also observed in [13] for the CSP). In addition, we mention that the model is also valid if equality is required in (8), but preliminary computational experiments showed that more instances are solved when we use inequalities.

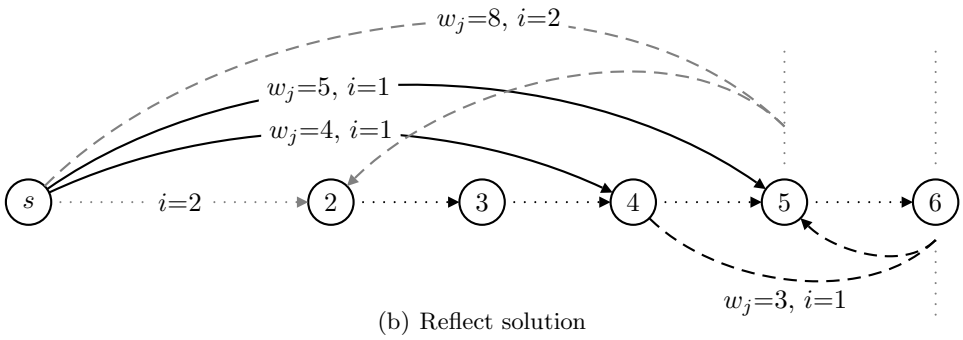
The main drawback of the Arc-flow model (6)-(11) is the huge amount of variables and constraints that it requires when large capacities are involved. A number of recent studies (see, e.g., Brandão and Pedroso [3], Clautiaux, Hanafi, Macedo, Voge, and Alves [9], Côté and Iori [12], and Delorme and Iori [13]) proposed techniques to overcome this difficulty. We decided to follow the footsteps of the approach proposed in [13] for bin packing and cutting stock problems.

4.2 Reflect model

The basic idea is to start from the Arc-flow model but to represent a knapsack using half of its capacity twice. More specifically, each arc whose head would lie in the second half of a knapsack is *reflected* into the first half. This is obtained by having, for each knapsack i, *standard item arcs* (arcs of the Arc-flow model terminating up to node $c_i/2$)) and *reflected item arcs* (mirrored at $c_i/2$ so that they terminate at a node in $\{0, \ldots, c_i/2\}$). The latter represent the occupation of the first half of knapsack i plus the remaining available space in the second half of the knapsack. To clarify this concept let us use again the example of the previous section (Figure 1). The way in which we deal with odd bin sizes and knapsack arcs is clarified after the example. Consider item 1 and its associated arc (0,8,1,0). For knapsack 1 we have $c_1/2 = 6 < e$, so this arc would terminate in the



(a) Reflect graph



(b) Reflect solution

Figure 2: Reflect graph and optimal solution for the instance of Figure 1

.

second half of the knapsack. The two units of the item occupying the second part of the knapsack are instead "mirrored" in the first part, producing an arc that starts at 0, "bounces" at 6, and terminates at 4, as shown in Figure 2-(a). For knapsack 2 we have $c_2/2 = 5$, so the reflected arc starts at 0 and terminates at 2.

Since we only model one half of the knapsack, a feasible solution in which some items fill a knapsack for more than half of its capacity must be represented by two paths starting from 0 and terminating in the same node, say e, with $e \leq c_i/2$. Exactly one among the two paths includes a reflected arc. Consider, e.g., the packing of items 1 and 3 into knapsack 1. This solution is represented, in Figure 2-(a), by the standard path going from 0 to 4 (indeed a single arc, with label '$w_j = 4$', in the figure), together with a second path consisting of the reflected arc going from 0 to 4 (dashed line with label '$w_j = 8$, i=1', in the figure). If we pack items 1 and 4 into knapsack 1 we represent the solution with two paths, one made by the same reflected arc from 0 to 4, and one made by the standard arc from 0 to 3, plus the loss arc from 3 to 4. The optimal solution, in which items 2, 3, and 4 are packed together again in the knapsack of capacity 12 and item 1 in the knapsack of capacity 10 is depicted in Figure 2-(b).

Formally, our Reflect model is built upon a multigraph $\mathcal{G}' = (\mathcal{N}', \mathcal{A}')$. Suppose for the sake of simplicity that all c_i values are even, let c be the maximum c_i value, and let $\mathcal{N}' = \{s \equiv 0, 1, \ldots, c/2\}$ denote the node set. Similarly to Arc-flow, we use (d, e, j, i) to denote a generic arc belonging to $\mathcal{A}'$, going from node d to node e, possibly involving item j and knapsack i. This time we distinguish among four subsets of arcs, partitioning $\mathcal{A}'$ into ${\mathcal{A}'}_s \cup {\mathcal{A}'}_r \cup {\mathcal{A}'}_c \cup {\mathcal{A}'}_l$ as follows:

- ${\mathcal{A}'}_s = \{(d, d + w_j, j, 0)$ with $j \in \{1, \ldots, n\}\}$ is the set of *standard item arcs* of the Arc-flow model;
- ${\mathcal{A}'}_r = \{((d, c_i - (d + w_j), j, i)$ with $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}\}$ is the set of *reflected item arcs* (satisfying $d + w_j > c_i/2$ and $d \leq c_i - w_j$);
- ${\mathcal{A}'}_c = \{(c_i/2, c_i/2, 0, i)$ with $i \in \{1, \ldots, m\}\}$ is the set of *reflected connection arcs*;
- ${\mathcal{A}'}_l = \{(d, e, 0, 0)\text{with}d < e\}$ is the set of *loss arcs* as in the original Arc-flow model.

Both standard and reflected item arcs are used to represent the packing of an item j at a certain knapsack filling d. Reflected connection arcs are used to merge two sub-paths, both ending at $c_i/2$, into a unique path, whereas loss arcs model an empty space between d and e. (For the sake of clarity reflected connection arcs are not shown in Figure 2.)

It has been proved in [13] that the number of arcs in the Reflect model can be reduced through two properties:

(a) given a knapsack i, the mirroring can be restricted to arcs with $d < c_i/2$, avoiding the generation of all arcs having the tail in the second half of the knapsack;

(b) given a knapsack i and an item j, the mirroring can be restricted to Arc-flow arcs with $c_i - (d + w_j) < d$, i.e., the generation of reflected arcs with tail, d, greater than head, e, can be avoided.

The rationale of the two properties is that a solution involving the suppressed arcs can always be represented by the maintained arcs.

The procedure used to construct sets $\mathcal{N}'$ and $\mathcal{A}'$ is provided in Algorithm 2. Preliminarily sorting the items by non-increasing weight is computationally convenient for this algorithm too. The usual array T keeps track of the possible arc tails. For each item j (Step 5), it scans all possible tails l that could lead to a valid arc (Step 6). First, it checks if a standard arc can be

created (i.e., $l + w_j \leq c/2$) (Steps 8-11). Then, it tries to create a reflected arc for each knapsack i (Steps 12-17). An arc can be reflected in $c_i/2$ if its head ends after $c_i/2$ and it satisfies the above reduction properties. The algorithm terminates by adding knapsack and loss arcs (Steps 22 and 23).

Algorithm 2 `Create_reflect_multigraph_MKP`

1: **Input:** m: number of knapsacks; c_i: knapsack capacities; n: number of items; w_j: item weights
2: $\mathcal{N}' \leftarrow \emptyset$; $\mathcal{A}'_s \leftarrow \emptyset$; $\mathcal{A}'_r \leftarrow \emptyset$; $\mathcal{A}'_c \leftarrow \emptyset$; $\mathcal{A}'_l \leftarrow \emptyset$; $\mathcal{A}' \leftarrow \emptyset$; $s \leftarrow 0$; $c \leftarrow \max_{i=1,\ldots,m}\{c_i\}$
3: $T[l] \leftarrow 0$ **for** $l = 1, \ldots, \frac{c}{2}$ ▷ an array that keeps track of the possible tails
4: $T[s] \leftarrow 1$; $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{s\}$ ▷ node s is the first tail
5: **for** $j = 1$ **to** n **do** ▷ item arcs
6: **for** $l = \frac{c}{2} - 1$ **down to** 0 **do**
7: **if** $T[l] = 1$ **then** ▷ if l is a possible tail
8: **if** $l + w_j \leq \frac{c}{2}$ **then** ▷ standard item arcs
9: $\mathcal{A}'_s \leftarrow \mathcal{A}'_s \cup \{(l, l + w_j, j, 0)\}$
10: $T[l + w_j] \leftarrow 1$; $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{(l + w_j)\}$ ▷ also added to the possible tails
11: **end if**
12: **for** $i = 1$ **to** m **do** ▷ reflected item arcs
13: **if** $l + w_j > \frac{c_i}{2}$ **and** $l \leq c_i - (l + w_j)$ **then** ▷ reflection in $\frac{c_i}{2}$
14: $\mathcal{A}'_r \leftarrow \mathcal{A}'_r \cup \{(l, c_i - (l + w_j), j, i)\}$
15: $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{(c_i - (l + w_i))\}$ ▷ not added to the possible tails
16: **end if**
17: **end for**
18: **end if**
19: **end for**
20: **end for**
21: **for** $i = 1$ **to** m **do** $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{\frac{c_i}{2}\}$
22: **for** $i = 1$ **to** m **do** $\mathcal{A}'_c \leftarrow \mathcal{A}'_c \cup \{(\frac{c_i}{2}, \frac{c_i}{2}, 0, i)\}$ ▷ reflected connection arcs
23: **for** $l \in \mathcal{N}'$ **do** $\mathcal{A}'_l \leftarrow \mathcal{A}'_l \cup \{(l, l', 0, 0)\colon\ l' = \min(e \in \mathcal{N}' : e > l)\}$ ▷ loss arcs
24: $\mathcal{A}' \leftarrow \mathcal{A}'_s \cup \mathcal{A}'_r \cup \mathcal{A}'_c \cup \mathcal{A}'_l$
25: **return** $\mathcal{N}', \mathcal{A}'$

The graph created by Algorithm 2 for the instance of Figure 1 is given in Figure 2-(a).

Let again $\delta^+(e)$ and $\delta^-(e)$ denote the subsets of arcs emanating from and entering node e. Let ξ_{deji} be an integer variable giving the number of times arc (d, e, j, i) is chosen, and let t_j be a binary variable taking the value 1 iff item j is selected. By using r instead of i when we refer to a reflected arc, the MKP can be modeled as:

$$\max \quad \sum_{j=1}^{n} p_j t_j \tag{12}$$

$$\text{s.t.} \quad \sum_{(d,e,j,0)\in\delta^-(e)\cap\mathcal{A}'_s} \xi_{dej0} = \sum_{\substack{(d,e,j,r)\in\delta^-(e)\cap\mathcal{A}'_r \\ r>0}} \xi_{dejr} + \sum_{(e,f,j,i)\in\delta^+(e)} \xi_{efji} \quad e \in \mathcal{N}' \setminus \{0\}, \tag{13}$$

$$\sum_{(0,e,j,i)\in\delta^+(0)} \xi_{0eji} + \sum_{\substack{(0,0,j,r)\in\delta^-(0) \\ r>0}} \xi_{00jr} = 2m, \tag{14}$$

$$\sum_{(d,e,j,i)\in \mathcal{A'}_s \cup \mathcal{A'}_r} \xi_{deji} \geq t_j \qquad j = 1,\ldots,n, \tag{15}$$

$$\sum_{(d,e,j,i)\in \mathcal{A'}_c} \xi_{deji} \leq 1 \qquad i = 1,\ldots,m, \tag{16}$$

$$\xi_{deji} \in \mathbb{N} \qquad (d,e,j,i) \in \mathcal{A'}_l, \tag{17}$$

$$\xi_{deji} \in \{0,1\} \qquad (d,e,j,i) \in \mathcal{A'}_s \cup \mathcal{A'}_r, \cup \mathcal{A'}_c \tag{18}$$

$$t_j \in \{0,1\} \qquad j = 1,\ldots,n. \tag{19}$$

Objective function (12) maximizes the total profit. Constraints (13) ensure that the flow on standard arcs entering a node e is equal to the flow (on both standard and reflected arcs) emanating from e, plus the flow on reflected arcs entering e. Constraint (14) forces the amount of flow emanating from 0 to be twice the number of knapsacks used, and additionally takes into account the fact that a reflected arc can also directly enter node 0 (in case there are an item j and a knapsack i such that $w_j = c_i$). Constraints (15) link together variables t_j and ξ_{deij}, while constraints (16) impose that at most one reflected arc per knapsack is selected, i.e., that each knapsack is used at most once.

We conclude with two observations. We supposed so far that all c_i values are even, so as to maintain integer indices for the $c_i/2$ nodes. Odd values can be handled by multiplying weights and capacities by 2, thus leaving the number of variables and constraints unchanged.

The second observation is that model (12)-(19) can be strengthened by adding the constraint

$$\sum_{j=1}^{n} t_j \leq n_{\max}, \tag{20}$$

where $n_{\max}$ is the maximum number of items that can be inserted into the knapsacks. Determining the exact value of $n_{\max}$ is an $\mathcal{NP}$-hard problem, as it requires solving a variable-sized extension of the maximum cardinality bin packing problem (see Section 1). In practice, $n_{\max}$ can be obtained by preliminarily setting all item profits to 1 and solving (12)-(19) with a given time limit. If an optimal solution is not reached, an upper bound on $n_{\max}$ can be still obtained by rounding-down the upper bound provided by the MIP solver when the time limit is reached. Note that such simple improvement can also be adopted for the classical model (1)-(4) and for the Arc-flow model (6)-(11).

5 Decomposition methods

Despite the fact that Reflect produces a sharp decrease in the number of variables and constraints with respect to Arc-flow, this number can still be too big for large-size instances. In this section, we present two decomposition techniques that can further reduce it.

5.1 Knapsack-based decomposition

A classical MKP decomposition consists in: (i) solving the surrogate dual relaxation of the problem, that is given (see [39]) by a single knapsack problem having a unique knapsack of capacity $C = \sum_{i=1}^{m} c_i$, and then (ii) checking if it is possible to partition the selected items among the m knapsacks. The second phase either provides a feasible (and hence optimal) solution or a cut that can be added to the surrogate problem. The *Master Problem* (MP) of this decomposition is

$$\max \quad z = \sum_{j=1}^{n} p_j \; t_j \tag{21}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j \; t_j \leq C, \tag{22}$$

$$t_j \in \{0,1\} \qquad j = 1, \ldots, n. \tag{23}$$

Let $\bar{t_j}$ be the solution of (21)–(23). The second stage *Slave Problem* (SP) is to look for a feasible partition of the items in $\bar{J} = \{j \in 1, \ldots, n : \bar{t_j} = 1\}$. We first invoke the constraint programming (CP) algorithm introduced in Section 1. If this fails in either finding a feasible solution, or in proving that none exists, we attempt to solve the SP with a reduced version of the Reflect model (12)–(19) obtained by restricting the item set to $\bar{J}$, and by disregarding the objective function (in practice, constraints (13)-(17) with (15) replaced by $\sum_{(d,e,j,i)\in\mathcal{A}'} \xi_{deji} \geq 1, j \in \bar{J}$). If we obtain a feasible solution, this is optimal for the MP. Otherwise, we add the *no-good cut*

$$\sum_{j\in\bar{J}} t_j \leq |\bar{J}| - 1, \tag{24}$$

to the MP and we iterate.

We propose two methods for improving this decomposition. As constraints (24) are known to be weak in practice, the first method improves them by finding the minimal subset $J' \subseteq \bar{J}$ such that the SP remains infeasible. We start with $J' = \bar{J}$, iteratively remove the smallest item in J', and stop as soon as the SP becomes feasible. The last J' that produced infeasibility is then used to create the resulting *combinatorial Benders' cut* (see, e.g., Codato and Fischetti [10])

$$\sum_{j\in J'} t_j \leq |J'| - 1, \tag{25}$$

which is used to replace (24). Worth is mentioning that the use of cuts of type (25) has led to good results in a number of cutting and packing problems (see, e.g., Côté et al. [11] and Delorme et al. [15]). A similar idea, although not expressed in terms of combinatorial Benders' cuts but still based on the search for a smallest infeasible subset, was used by Pisinger and Sigurd [45] for the two-dimensional bin packing problem.

The second improvement is based on the consideration that the proposed decomposition has to find MP integer solutions before being able to generate the necessary cuts that will possibly lead to a proven optimal solution. We start by generating standard Benders' infeasibility cuts. We solve the Linear Programming (LP) relaxation of the MP (21)-(23), obtaining a solution $\bar{t_j}$ and a set $\bar{J} = \{j \in 1, \ldots, n : \bar{t}_j > 0\}$ of items that are entirely or partially selected. We then check the feasibility of the LP relaxation of a reduced version of the Reflect model made by (13)-(17) with (15) replaced by $\sum_{(d,e,j,i)\in\mathcal{A}'} \xi_{deji} \geq \bar{t_j}, j \in \bar{J}$.

Consider now the dual variables α (associated with constraint (14)), β_j (associated with constraints (15)), and γ_i (associated with constraints (16)). If the LP relaxation is proved to be infeasible, then the following Benders' feasibility cut is added to the MP

$$2m\alpha + \sum_{j\in\bar{J}} \beta_j t_j + \sum_{i=1}^{m} \gamma_i \geq 0, \tag{26}$$

and the process is iterated. If instead a feasible solution is found for both LP relaxations (of the MP and the SP), then we invoke again the original decomposition with the integrality constraints. The advantage of this approach lies in the generation of good initial cutting planes by means of (26).

5.2 Reflect-based decomposition

As the MP (21)–(23) does not incorporate any information about the different knapsack capacities, the decomposition algorithm of Section 5.1 may iterate many times before obtaining a proven optimal solution. We thus developed a second decomposition that works as the one of Section 5.1 but makes use of a more complex MP.

We use model (12)-(17) in which integrality constraints (17) are relaxed but constraints (19) are kept. Let again $\bar{t}_j$ $(j = 1, \ldots, n)$ denote the item variables in the solution of the relaxed model and let $\bar{J} = \{j \in 1, \ldots, n : \bar{t}_j = 1\}$ be the set of selected items. The solution can be unfeasible for the original problem because the arcs may have fractional values, implying that a selected item has different item fractions assigned to different knapsacks. The SP is the same used in the knapsack-based decomposition. Note that in this case we may use the Reflect model twice: the first time to select the items and the second time to check if they can be feasibly packed into the m knapsacks.

We still improve the decomposition through cuts (25), but we disregard the use of (26) as an LP solution for this MP is always feasible for the LP relaxation of the corresponding SP.

6 Overall algorithm

On the basis of preliminary computational experiments, we developed an effective hybrid algorithm that invokes in sequence the main components presented in the previous sections. The overall procedure, called *Hybrid MKP* (Hy-MKP), is presented in Algorithm 3, and, as shown in the next section, is a powerful exact method that achieves proven optimal solutions for many benchmark instances.

We initially apply, at Step 1, the preprocessing techniques of Section 3. This phase is followed by the execution (Step 2) of the MULKNAP branch-and-bound algorithm by Pisinger [44] (see Section 2) for a short amount, τ, of CPU seconds: this can allow a quick solution of easy instances and of instances with a high n/m ratio. If an optimal solution is not obtained, Step 3 constructs the reflect multigraph through the method presented in Section 4.2. At Steps 4-8 we then perform, for at most ν times, the knapsack-based decomposition of Section 5.1 with only the no-good cut (24), i.e., without (25) and (26): indeed, combinatorial Benders' cuts and Benders' feasibility cuts tend to slow down the overall process, and our strategy is to use this decomposition to solve instances that require few cuts. The remaining instances are solved at Step 9, where we fully execute the Reflect-based decomposition of Section 5.2 of the current MP (i.e., including all cuts obtained at Step 7).

Algorithm 3 `Hy-MKP`

1: perform *Instance reduction*, *Capacity lifting*, and *Item dominance*;
2: call MULKNAP for τ seconds; **if** the solution is optimal **then return**;
3: call Create_reflect_multigraph_MKP;
4: **for** $i := 1$ **to** ν **do**
5: execute the knapsack-based decomposition;
6: **if** an optimal solution has been obtained **then return**
7: **else** add the resulting no-good-cut
8: **end for**;
9: **if** the instance is not solved **then** execute the Reflect-based decomposition;
10: **return**.

7 Computational experiments

We report in this section the outcome of extensive computational experiments aimed at testing the effectiveness of the proposed approaches.

Benchmarks

Each algorithm was run on 5 benchmark sets:

- SMALL: a set of 180 instances (with $m \in \{10, 20\}$ and $n \in \{20, 40, 60\}$) proposed by Kataoka and Yamada [32] for a variant of the MKP in which the items are partitioned into subsets and an additional constraint imposes that a knapsack can only contain items of the same class. We adapted the instances to the MKP by simply disregarding the additional constraint. All the SMALL instances that we used have been kindly provided by the authors. Other instances proposed in [32], with $m \in \{200, 400, 800\}$ and $n \in \{4000, 8000\}$, are too large to be solved when adapted to the MKP. We thus obtained larger instances as follows.

- FK_1, FK_2, FK_3, and FK_4: four sets of 480 larger instances each, that reproduce those that Fukunaga [23] derived from the classical benchmarks in Chapter 6 of [42] with the aim of identifying critical ratios n/m producing difficult instances. The instances were obtained through Pisinger's instance generator (available online at `http://www.diku.dk/~pisinger/codes.html`) .

All test sets are available at `http://or.dei.unibo.it/library`. In Table 1, we better detail their characteristics. Set SMALL contains 18 ($2 \times 3 \times 3$) subsets of 10 instances each, obtained with all the combinations of m, n and 'class of correlation' (to be defined below). Each set FK contains 24 (6×4) subsets of 20 instances each, created with the objective of studying the impact of six different ratios n/m (2, 3, 4, 5, 6, and 10) on the correlation classes. The table reports, for each instance set, the values of m and n and the classes of correlation, defined by Kataoka and Yamada [32] as follows. The weights w_j are always uniformly distributed in $[\alpha, 1000]$, with $\alpha = 1$ for the SMALL instances and $\alpha = 10$ for the FK instances. The four *correlation classes* are:

- *uncorrelated*: profits p_j are uniformly distributed in $[\alpha, 1000]$;

- *weakly correlated*: For SMALL, $p_j = 0.6w_j + \vartheta_j$, with ϑ_j uniformly random in $[1, 400]$. For FK, the p_j values are uniformly distributed in $[w_j - 100, w_j + 100]$, such that $p_j \geq 1$;

- *strongly correlated*: For SMALL, $p_j = w_j + 200$, for FK, $p_j = w_j + 10$;

- *subset-sum*: $p_j = w_j$.

For the SMALL instances, the knapsack capacities were generated as $c_i = \lfloor \sigma \lambda_i \sum_{j=1}^{n} w_j \rfloor$, with λ_i uniformly distributed in $[0, 1]$ such that $\sum_{i=1}^{m} \lambda_i = 1$, and $\sigma \in \{0.25, 0.50, 0.75\}$. For the FK instances, we generated so called *similar capacities* (see [42]): c_i uniformly random in $[0.4 \sum_{j=1}^{n} w_j/m, 0.6 \sum_{j=1}^{n} w_j/m]$ for $i = 1, \ldots, m-1$, and $c_m = 0.5 \sum_{j=1}^{n} w_j - \sum_{i=1}^{m-1} c_i$. Instances with $\min_j\{w_j\} > \min_i\{c_i\}$, $\max_j\{w_j\} > \max_i\{c_i\}$, or $\sum_{j=1}^{n} w_j \leq \max_i\{c_i\}$, were discarded and generated again.

Implementation details

For the instance reduction procedure of Section 1, the CP optimizer was invoked at each iteration (if required) with a time limit of one second, terminating the reduction procedure as soon as it could find no solution.

Table 1: Characteristics of the 5 sets of instances

set	n/m	item weights	correlation between profits and weights
SMALL	{20/10, 40/10, 60/10, 20/20, 40/20, 60/20}	[1–1000]	{uncorrelated, weekly, strongly}
FK_1	{60/30, 45/15, 48/12, 75/15, 60/10, 100/10}	[10–1000]	{uncorrelated, weekly, strongly, subset-sum}
FK_2	{120/60, 90/30, 96/24, 150/30, 120/20, 200/20}	[10–1000]	{uncorrelated, weekly, strongly, subset-sum}
FK_3	{180/90, 135/45, 144/36, 225/45, 180/30, 300/30}	[10–1000]	{uncorrelated, weekly, strongly, subset-sum}
FK_4	{300/150, 225/75, 240/60, 375/75, 300/50, 500/50}	[10–1000]	{uncorrelated, weekly, strongly, subset-sum}

The upper bound $n_{\max}$ on the number of items that can be inserted into the knapsacks (see Section 4.2) was obtained by solving the corresponding model (i.e., (1)-(4) for the Classical model, (6)-(11) for the Arcflow model, and (12)-(19) for the other approaches) with all profits set to 1. If the exact value was not found after 120 seconds, the execution was terminated and the rounded down value of the upper bound returned by the MIP solver was used.

The decomposition approaches of Section 5 were implemented by executing the Gurobi MIP solver on the master problem, and using the incumbent callback functions to invoke it for the slave problems. The callback is activated by the solver as soon as it finds a feasible integer solution, and the slave problem is solved on such solution: the resulting cut is added to the master, and the control is resumed by the solver.

For the knapsack-based decomposition called at Step 5 of Hy-MKP (see Section 6), as at most ν cuts are allowed, we solved to optimality the master problem, before trying to solve a slave problem.

The slave problems were normally attacked by first trying the CP approach for one second and then, if it fails, switching to the Reflect model. However, for instances where Reflect was very unlikely to solve the problem because of the large amount of variables involved by the model (threshold set at $c_i > 10^7$), only the CP approach was executed, without any local time limit.

The two parameters of Hy-MKP were established as $\tau = 2$ and $\nu = 10$. These values were determined by some preliminary experiments and aimed at getting the best compromise between the amount of time spent in each component and the number of optimal solutions they found.

Results

All algorithms were coded in C++. The experiments were performed on an Intel Xeon E5530, 2.4 GHz with 24 GB of memory, running under Linux Ubuntu 14.04 LTS 64-bit, using a single core. We used Gurobi 6.5.1 to solve the MIP models and Cplex 12.6.2 to solve the CP subproblems (as Gurobi does not have a CP solver).

Table 2 compares the proposed methods with the best exact approaches from the literature on benchmarks SMALL and FK. The C implementation of the branch-and-bound code MULKNAP by Pisinger [44] is available at `http://www.diku.dk/~pisinger/codes.html`. We contacted the author of [23] to ask for his FK_1 instances, but unfortunately these are not available anymore. For the sake of comparison we report the results given in [23], obtained on an Intel Xeon E5440, 2.83 GHz, with a time limit of one hour. (The instances tested in [23] were generated exactly as our FK_1 instances, although, of course, they cannot be considered identical to ours.)

Column "method" describes the approach, with some attribute describing the specific implementation. The "Classical model" is (1)-(4), solved with the MIP solver. Attribute "t_j" indicates the reformulation obtained by adding the item selection binary variables t_j described in Section 3. Attribute "priority" means that we use the solver option to first select t_j variables for branching in the branch-and-cut process. "Arc-flow model - t_j" and "Reflect model - t_j" are (6)–(11) and (12)–(19) of Section 4, without using t_j variables. Attribute "+ n_{max}" denotes the use of constraint

Table 2: Comparison of the proposed methods for instance sets SMALL and FK_1

Method	SMALL		FK_1	
	#opt	time	#opt	time
MULKNAP	150	229.4	353	104.4
2D/PS +B[(1)]	–	–	459	292.5
Classical	115	483.7	217	722.2
Classical + t_j	131	382.5	215	715.4
Classical + t_j + Priority	131	387.4	197	760.1
Classical + t_j + n_{max}	139	329.1	220	719.0
Arcflow	120	434.1	298	499.0
Arcflow + t_j	141	284.8	326	469.4
Arcflow + t_j + Priority	155	222.5	358	403.0
Arcflow + t_j + Priority + n_{max}	169	138.5	383	341.1
Reflect	141	309.3	310	460.7
Reflect + t_j	149	240.7	355	383.0
Reflect + t_j + Priority	164	146.6	388	295.2
Reflect + t_j + Priority + n_{max}	179	53.6	423	223.1
Knapsack-based decomposition + (25)	166	131.1	394	277.4
Knapsack-based decomposition + (25) + (26)	173	72.3	420	215.6
Reflect-based decomposition + (25)	**180**	15.7	477	83.4
Hy-MKP	**180**	11.5	**480**	10.3
– Preprocessing & MULKNAP	121	0.9	271	1.0
– Knapsack-based decomp.	15	8.1	99	6.4
– Reflect-based decomp.	44	31.4	110	25.0

[(1)] Values from Fukunaga [23] on instances generated by the author, times on an Intel Xeon E5440 2.83 GHz

(20) to limit the sum of the t_j variables. "Knapsack-based decomposition" and "Reflect-based decomposition" are the methods of Sections 5.1 and 5.2, respectively, implemented with cuts (25). The possible use of improved Benders' cuts is identified by attribute "+ (26)". "Hy-MKP" is the overall exact decomposition algorithm of Section 6. For the latter algorithm we also report the performance of each component: preprocessing & MULKNAP, Knapsack-based decomposition, and Reflect-based decomposition.

Each algorithm was given an overall time limit of 1200 seconds. Columns "#opt" give the number of optimal solutions found, while columns "time" report the average CPU time computed over all runs, including the ones terminated by the time limit. For "Hy-MKP", "#opt" gives the number of instances closed by the corresponding component, and "time" reports the average CPU time used by the component, if called. For Classical, Arc-flow, and Reflect models, we also provide in Table 3 some information about the average continuous relaxation value and time in columns "LP rel. value" and "LP rel. time". Columns "gap" and "max gap" give, respectively, the average and maximum percentage gap. The percentage gap is computed as $100(UB_A - UB^*)/UB^*$, where UB_A is the value of the continuous relaxation of model A, and UB^* is the optimal solution value. We also report in columns "# var." and "# cons." the average number of variables and constraints required by the models, respectively. Note that priorities do not have any impact on the model size or on the continuous relaxation. Tables 2 and 3 show a number of interesting facts:

- even if it is nearly 20 years old, MULKNAP is still competitive: It solved 150 SMALL instances out of 180 and 353 FK_1 instances out of 480;
- the branch-and-bound by Fukunaga [23], in its 2D/PS+B configuration, is quite effective in finding proven optimal solutions (459 out of 480 instances in less than 300 seconds on

Table 3: Model characteristics of the proposed methods for instance sets SMALL and FK_1

Method	SMALL						FK_1					
	LP rel. value	LP rel. time	gap	max gap	# var.	# cons.	LP rel. value	LP rel. time	gap	max gap	# var.	# cons.
Classical	15994.0	0.0	1.9	17.4	561	286	18464.1	0.0	0.4	5.5	672	355
Classical + t_j	15994.0	0.0	1.9	17.4	603	286	18464.1	0.0	0.4	5.5	731	355
Classical + t_j + n_{max}	15918.2	0.0	1.5	17.4	603	287	18452.0	0.0	0.3	5.5	731	356
Arcflow	15829.0	5.9	0.3	3.6	17999	1448	18420.8	14.3	0.1	0.9	34701	1689
Arcflow + t_j	15829.0	2.8	0.3	3.6	18041	1448	18420.8	6.1	0.1	0.9	34760	1689
Arcflow + t_j + Priority + n_{max}	15790.1	2.8	0.2	2.2	18041	1449	18419.8	5.2	0.1	0.9	34760	1690
Reflect	15832.9	1.6	0.4	3.6	7167	865	18421.2	4.7	0.1	0.9	19946	1005
Reflect + t_j	15832.9	0.6	0.4	3.6	7209	865	18421.2	1.9	0.1	0.9	20005	1005
Reflect + t_j + Priority + n_{max}	15792.3	0.7	0.2	2.2	7209	866	18420.1	1.7	0.1	0.9	20005	1006

The average optimal solution for SMALL (resp. FK_1) is 15765.2 (resp. 18400.8)

average);

- Arc-flow and Reflect compare favorably with the classical model, solving more instances in smaller computing times, and with better continuous relaxation values;
- the use of variables t_j improves on the performances both in terms of optimal solutions and of computing time. It also significantly decreases the time spent in solving the LP relaxation of Arc-flow and Reflect. A possible explanation is that only n t_j variables are required in the objective function of Arc-flow, while $|\mathcal{A}_s|$ x_{deji} variables are needed in the original version;
- giving priority to the t_j variables in the branching process further improves on the performance for all models but the classical one. This is probably explained by the fact that the information obtained after branching on a t_j variable are more "balanced" than those obtained by branching on another variable. For example, setting x_{deji} to 1 in Arc-flow forces all other item arcs associated with j to take the value 0 and incorporates the profit of j in the objective function, while no reduction is deduced from setting x_{deji} to 0. Conversely, setting t_j to 0 forces all item arcs associated with j to take the value 0, while setting t_j to 1 incorporates the profit of j in the objective function;
- the use of inequality (20) gives an additional advantage both in terms of average CPU time and LP relaxation value;
- the knapsack-based decomposition is competitive with the flow-based approaches, especially when coupled with both improved cuts;
- the Reflect-based decomposition with combinatorial Benders' cuts is extremely fast and can solve all SMALL instances and all but 3 FK_1 instances. However, it did not dominate the other methods as the 3 unsolved FK_1 instances could be solved to proven optimality by the knapsack-based decomposition in 129.2 seconds on average;
- the combined decomposition algorithm Hy-MKP was the best approach and solved to optimality all instances in 11 seconds, on average. All of its components appear to be useful: more than half the instances are closed by MULKNAP, and the remaining instances are either solved by the knapsack-based decomposition (that runs for a bit more than 10 seconds on average, when called) or by the Reflect-based decomposition.

The performance of the proposed methods on larger instances FK_2, FK_3, and FK_4 was analyzed for the most competitive version of each algorithm. (Results for FK_1 are added for the sake of comparison.) In this case we could not evaluate the branch-and-bound algorithm by Fukunaga [23], since it is not available and these instances were not tested in [23]. The outcome of these experiments is reported in Table 4, for which we can observe the following:

- MULKNAP is very resilient to size increase;
- the classical model behaves poorly for larger instances;
- the behavior of Arc-flow and Reflect (in all configurations) constantly worsen when the number of items and knapsacks increases, and their performance for FK_4 is very close to that of the classical model;
- the combined algorithm Hy-MKP was definitely the best approach, although its performance deteriorates when the size increases. While the MULKNAP component seems to benefit from size increase, the opposite holds for the (pseudo-polynomial) Reflect-based decomposition component.

To get a better understanding of the good computational performance of algorithm Hy-MKP, we provide in Table 5 the detailed results obtained by each of the selected algorithms for instance sets FK_2. Each row corresponds to a set of 20 instances. Rows are reported in the table by increasing value of the n/m ratio, shown in the first column. Columns "#opt" report again the number of proven optimal solutions out of 20. Columns "time" report the average CPU time computed over all runs, including the ones terminated by the time limit. Columns "gap" give the percentage gap, computed as $100(LB^* - LB_A)/LB^*$, where LB_A is the value of the best feasible solution value provided by algorithm A, and LB^* is the optimal solution value (with the exception of a single instance, with $n/m = 3$ and "subset-sum" correlation, where no method produced a proven optimal solution, so we set LB^* to the best solution value produced by all algorithms).

MULKNAP is very efficient when either $n/m \in \{2, 10\}$ or $n/m \in \{5, 6\}$ and the instances have "strongly" or "subset-sum" correlation. In the other cases its performance is generally poor, and it is extremely bad for $n/m = 3$. This confirms an observation made in [23]: MKP instances that are hard for branch-and-bound algorithms have a ratio n/m between 3 and 4. It is also worth observing that the percentage gap of the non-optimal feasible solutions provided by MULKNAP is generally very small, never reaching, on average, 2%.

Table 4: Comparison of the most competitive version of each method for instance sets FK

Method	FK_1		FK_2		FK_3		FK_4	
	#opt	time	#opt	time	#opt	time	#opt	time
MULKNAP	353	104.4	290	461.1	311	449.3	313	441.9
Classical model $+t_j$ $+n_{max}$	220	719.0	90	984.8	80	1001.8	80	1003.8
Arc-flow model +priority $+n_{max}$	383	341.1	276	614.7	179	802.0	96	986.6
Reflect model +priority $+n_{max}$	423	223.1	282	576.0	188	650.5	97	986.2
Reflect-based decomposition +(25)	477	83.4	338	531.1	228	596.5	111	976.4
Hy-MKP	**480**	10.3	**469**	91.9	**461**	146.1	**398**	286.9
–Preprocessing & MULKNAP[(1)]	271	1.0	279	1.4	306	1.8	311	2.7
–Knapsack-based decomposition	99	6.4	103	43.5	96	195.5	65	365.9
–Reflect-based decomposition	110	25.0	87	318.3	59	357.5	22	541.8

[(1)] MULKNAP exceeded its 2 seconds time limit for instances with ratio $n/m = 2$

Table 5: Detailed comparison of the most competitive methods for instance FK_2

Instances			MULKNAP			Classical model $+t_j+n_{max}$			Arc-flow model +priority $+n_{max}$			Reflect model +priority $+n_{max}$			Reflect-based decomposition +(25)			Hy-MKP		
ratio	n/m	correlation	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time
2	120/60	uncorrelated	20	0.00	3.1	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0
		weakly	20	0.00	3.2	20	0.00	3.2	20	0.00	3.2	20	0.00	3.2	20	0.00	3.2	20	0.00	3.2
		strongly	20	0.00	6.5	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.2
		subset-sum	20	0.00	3.6	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.0	20	0.00	3.2
3	90/30	uncorrelated	0	1.96	t.l.	0	0.16	t.l.	20	0.00	38.2	19	0.00	83.4	20	0.00	21.6	20	0.00	24.9
		weakly	0	1.68	t.l.	9	0.07	857.4	20	0.00	26.7	20	0.00	35.9	20	0.00	25.1	20	0.00	27.7
		strongly	0	0.92	t.l.	0	0.79	t.l.	18	0.00	479.5	20	0.00	246.6	16	0.00	631.4	16	0.00	684.8
		subset-sum	0	0.37	t.l.	0	0.37	t.l.	19	0.00	156.3	18	0.00	180.3	10	0.00	880.5	13	0.00	781.7
4	96/24	uncorrelated	0	1.76	t.l.	0	0.45	t.l.	17	0.01	270.0	19	0.00	120.0	20	0.00	70.9	20	0.00	26.9
		weakly	0	1.76	t.l.	0	0.48	t.l.	19	0.00	190.4	16	0.00	317.7	20	0.00	85.8	20	0.00	119.4
		strongly	1	0.40	1144.5	0	0.50	t.l.	13	0.02	859.9	16	0.01	574.0	20	0.00	301.1	20	0.00	46.8
		subset-sum	19	0.00	148.2	0	0.18	t.l.	19	0.00	186.5	19	0.00	117.2	20	0.00	298.8	20	0.00	115.1
5	150/30	uncorrelated	7	0.25	784.4	0	0.32	t.l.	13	0.02	672.2	14	0.02	603.5	19	0.00	496.0	20	0.00	126.2
		weakly	0	0.58	t.l.	0	0.76	t.l.	6	0.08	1032.0	3	0.06	1085.1	19	0.00	581.1	20	0.00	151.1
		strongly	20	0.00	0.0	0	0.37	t.l.	1	0.52	1173.9	0	0.10	t.l.	0	0.08	t.l.	20	0.00	0.0
		subset-sum	20	0.00	0.0	0	0.20	t.l.	8	0.08	880.5	6	0.07	969.4	3	0.07	1152.8	20	0.00	0.0
6	120/20	uncorrelated	19	0.00	60.3	1	0.18	1152.3	14	0.07	692.0	16	0.02	458.9	20	0.00	291.9	20	0.00	6.1
		weakly	4	0.12	970.1	0	0.55	t.l.	5	0.19	991.5	7	0.04	884.5	20	0.00	379.1	20	0.00	82.2
		strongly	20	0.00	0.0	0	0.24	t.l.	0	0.28	t.l.	0	0.06	t.l.	7	0.03	1026.2	20	0.00	0.0
		subset-sum	20	0.00	0.0	0	0.08	t.l.	4	0.17	1085.8	9	0.01	922.1	4	0.02	1151.6	20	0.00	0.0
10	200/20	uncorrelated	20	0.00	0.0	0	0.08	t.l.	0	11.11	t.l.	0	4.09	t.l.	13	0.02	717.5	20	0.00	0.0
		weakly	20	0.00	0.0	0	0.19	t.l.	0	7.83	t.l.	0	2.68	t.l.	0	5.10	t.l.	20	0.00	0.0
		strongly	20	0.00	0.0	0	0.08	t.l.	0	9.66	t.l.	0	5.86	t.l.	6	0.07	1022.0	20	0.00	0.0
		subset-sum	20	0.00	0.0	0	0.02	t.l.	0	3.91	t.l.	0	3.03	t.l.	1	0.05	1198.0	20	0.00	0.0

The classical model is very efficient for instances with $n/m = 2$, but it is not able to solve any of the other instances (but one). Its percentage gap is however always very small.

The performance of Arcflow and Reflect monotonically worsens when the ratio n/m increases. As observed by Delorme et al. [14], pseudo-polynomial formulations of cutting and packing problems exhibit poor performances when applied to instances with high capacities and relatively small items. In our case the knapsack capacities are directly related to the sum of the item weights, which increases with n/m. The feasible solutions provided when no optimal solution is found are generally very good, but for $n/m = 10$ where large percentage gaps are observed.

Hy-MKP solved to proven optimality all but eleven instances, all occurring for the ratio $n/m = 3$. This is not very surprising, as this ratio can be a critical one for packing problems. For example, a classical benchmark for the bin packing problem was obtained by Falkenauer [19] by generating random instances for which the optimal solution has three items per bin. Out of these eleven difficult instances, ten are solved to proven optimality by Arc-flow and Reflect. However, the feasible solutions provided by Hy-MKP for such ten instances are all optimal. For the remaining instance the gap is extremely small.

Hy-MKP takes advantage of its multiple components: when the branch-and-bound MULKNAP is able to solve the problem within few seconds, no further attempt is needed. When instead MULKNAP fails, the other components of Hy-MKP are able to very quickly find the optimal solution for almost all instances. For example, when $n/m = 3$, all weakly correlated instances were solved by the Reflect-based decomposition, while for $n/m = 5$ the weakly correlated instances were solved in one or two iterations by the knapsack-based decomposition.

All instances with ratio $n/m = 10$ were very quickly solved by the MULKNAP component of

Hy-MKP, while the other components behaved quite poorly. As observed by Pisinger [44], similar instances with much higher ratios (up to 10 000) can be solved by MULKNAP, so the same behavior has to be expected for Hy-MKP. For this reason, higher ratios were not tested. We also mention that, for the ILP models, we did not observe the frequently reported paradox that larger problems get easier, as in our case they involve more constraints and variables.

Summing up the results of our experiments, we can classify the instances as follows.

a. easy instances, with ratio $n/m = 2$, that are solved by almost all methods;

b. instances with ratio $n/m = 3$, that can be solved effectively (for reasonable size instances) by the Reflect-based decomposition;

c. instances with ratio $n/m = 4$, that can be solved effectively (for reasonable size instances) by the Reflect-based and knapsack-based decompositions;

d. instances with ratio $n/m = 5$, or 6, that can only be solved, in most cases, by the knapsack-based decomposition;

e. instances with high ratio n/m, for which branch-and-bound methods are particularly suited, while pseudo-polynomial approaches and decomposition methods are not.

To confirm these observations, we provide in Table 6 the same information as in Table 5, for instances of set FK_4. Symbol "–" in the gap columns indicates that no feasible solution was produced by the method. The instances appear to be much more difficult, but the algorithms' behavior does not change significantly.

Table 6: Detailed comparison of the most competitive methods for instances FK_4

Instances			MULKNAP			Classical model $+t_j+n_{max}$			Arc-flow model +priority $+n_{max}$			Reflect model +priority $+n_{max}$			Reflect-based decomposition +(25)			Hy-MKP		
ratio	n/m	correlation	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time	#opt	gap	time
2	300/150	uncorrelated	20	0.00	11.9	20	0.00	12	20	0.00	12.6	20	0.00	15.6	20	0.00	13.3	20	0.00	11.7
		weakly	20	0.00	11.8	20	0.00	11.9	20	0.00	13.3	20	0.00	12.6	20	0.00	12.7	20	0.00	11.7
		strongly	20	0.00	12.7	20	0.00	11.3	20	0.00	12.4	20	0.00	12.1	20	0.00	11.8	20	0.00	11.9
		subset-sum	20	0.00	11.4	20	0.00	11.3	20	0.00	12.0	20	0.00	11.7	20	0.00	11.6	20	0.00	11.9
3	225/75	uncorrelated	0	1.87	t.l.	0	0.76	t.l.	4	0.05	1066.1	4	0.09	1147.8	11	15.00	907.0	13	15.00	847.6
		weakly	0	1.35	t.l.	0	0.84	t.l.	5	0.01	1112.3	8	0.02	909.2	5	0.03	1098.1	7	0.03	1074.6
		strongly	0	0.46	t.l.	0	1.14	t.l.	1	0.39	1170.9	0	0.51	t.l.	0	0.11	t.l.	8	60.00	873.0
		subset-sum	0	0.20	t.l.	0	0.93	t.l.	0	0.18	t.l.	0	0.17	t.l.	0	0.10	t.l.	0	0.03	t.l.
4	240/60	uncorrelated	0	1.05	t.l.	0	0.45	t.l.	3	0.34	1104.5	4	0.17	1113.9	10	0.04	1005.6	17	14.55	429.1
		weakly	0	1.38	t.l.	0	1.07	t.l.	1	0.09	1174.1	1	0.15	1193.6	2	0.03	1166.0	8	39.48	907.0
		strongly	3	0.32	t.l.	0	0.53	t.l.	0	0.73	t.l.	0	0.49	t.l.	0	0.20	1214.3	16	20.00	507.1
		subset-sum	20	0.00	0.0	0	0.41	t.l.	2	0.20	1188.2	0	0.41	t.l.	3	0.20	1129.0	20	0.00	0.0
5	375/75	uncorrelated	18	0.00	t.l.	0	0.28	t.l.	0	9.25	t.l.	0	48.69	t.l.	0	30.10	t.l.	19	5.00	114.5
		weakly	0	0.19	t.l.	0	0.67	t.l.	0	17.39	t.l.	0	46.61	t.l.	0	25.07	t.l.	11	39.2	663.3
		strongly	20	0.00	0.0	0	0.42	t.l.	0	13.14	t.l.	0	46.42	t.l.	0	50.29	t.l.	20	0.00	0.0
		subset-sum	20	0.00	0.0	0	0.26	t.l.	0	10.56	t.l.	0	55.84	t.l.	0	70.15	t.l.	20	0.00	0.0
6	300/50	uncorrelated	20	0.00	0.0	0	0.18	t.l.	0	7.51	t.l.	0	5.85	t.l.	0	0.11	t.l.	20	0.00	0.0
		weakly	12	0.01	483.0	0	0.51	t.l.	0	4.52	t.l.	0	2.33	t.l.	0	5.12	t.l.	19	3.33	220.7
		strongly	20	0.00	0.2	0	0.28	t.l.	0	5.50	t.l.	0	7.34	t.l.	0	15.43	t.l.	20	0.00	0.0
		subset-sum	20	0.00	0.0	0	0.21	t.l.	0	5.97	t.l.	0	2.45	t.l.	0	25.35	t.l.	20	0.00	0.0
10	500/50	uncorrelated	20	0.00	0.0	0	0.05	t.l.	0	–	t.l.	0	–	t.l.	0	–	t.l.	20	0.00	0.0
		weakly	20	0.00	0.0	0	0.15	t.l.	0	–	t.l.	0	–	t.l.	0	–	t.l.	20	0.00	0.0
		strongly	20	0.00	1.1	0	0.10	t.l.	0	–	t.l.	0	–	t.l.	0	–	t.l.	20	0.00	1.0
		subset-sum	20	0.00	0.0	0	0.08	t.l.	0	–	t.l.	0	–	t.l.	0	–	t.l.	20	0.00	0.0

8 Conclusions

We have presented for the first time pseudo-polynomial formulations for the multiple knapsack problem, a well-known strongly $\mathcal{NP}$-hard combinatorial optimization problem with relevant managerial implications. We have embedded them into a novel exact method, that combines several techniques (branch-and-bound, Master/Slave decompositions, Benders'cuts, constraint programming). Extensive computational experiments have shown that the hybridization of different solution techniques can be successful in attacking an $\mathcal{NP}$-hard problem that is known to be very difficult to solve in practice.

Acknowledgments

Research supported by MIUR-Italy (Grant PRIN 2015, *Nonlinear and Combinatorial Aspects of Complex Networks*) and by Air Force Office of Scientific Research (under award number FA9550-17-1-0067). We thank three anonymous referees for helpful comments.

References

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

[2] M. Bergner and F.H.W. Dahms. Heterogeneous aggregation for Dantzig-Wolfe reformulation. Available at `http://www.or.rwth-aachen.de/research/publications/BergnerDahmsAggregation.pdf`, 2015.

[3] F. Brandão and J.P. Pedroso. Bin packing and related problems: General arc-flow formulation with graph compression. *Computers & Operations Research*, 69:56–67, 2016.

[4] H. Cambazard and B. O'Sullivan. Propagating the bin packing constraint using linear programming. In *Principles and Practice of Constraint Programming–CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 129–136. Springer-Verlag, 2010.

[5] A. Caprara, H. Kellerer, and U. Pferschy. The multiple subset-sum problem. *SIAM Journal on Optimization*, 11(2):308–319, 2000.

[6] A. Caprara, H. Kellerer, and U. Pferschy. A PTAS for the multiple subset-sum problem with different knapsack capacities. *Information Processing Letters*, 73(3):111–118, 2000.

[7] A. Caprara, H. Kellerer, and U. Pferschy. A 3/4-approximation algorithm for multiple subset sum. *Journal of Heuristics*, 9(2):99–111, 2003.

[8] C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, 2000.

[9] F. Clautiaux, S. Hanafi, R. Macedo, M.-E. Voge, and C. Alves. Iterative aggregation and disaggregation algorithm for pseudo-polynomial network flow models with side constraints. *European Journal of Operational Research*, 258:467–477, 2017.

[10] G. Codato and M. Fischetti. Combinatorial Benders' cuts for mixed-integer linear programming. *Operations Research*, 54(4):756–766, 2006.

[11] J.-F. Côté, M. Dell'Amico, and M. Iori. Combinatorial Benders' cuts for the strip packing problem. *Operations Research*, 62(3):643–661, 2014.

[12] J.-F. Côté and M. Iori. The meet-in-the-middle principle for cutting and packing problems. *INFORMS Journal on Computing*, 2018 (forthcoming).

[13] M. Delorme and M Iori. Enhanced pseudo-polynomial formulations for bin packing and cutting stock problems. Technical report, DEI "Guglielmo Marconi", Alma Mater Studiorum Università di Bologna, Italy, 2017 (submitted for publication). Available at `www.optimization-online.org/DB_FILE/2017/10/6270.pdf`.

[14] M. Delorme, M. Iori, and S. Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255:1–20, 2016.

[15] M. Delorme, M. Iori, and S. Martello. Logic based Benders' decomposition for orthogonal stock cutting problems. *Computers & Operations Research*, 78:290–298, 2017.

[16] H. Dyckhoff. A new linear programming approach to the cutting stock problem. *Operations Research*, 29:1092–1104, 1981.

[17] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

[18] S. Eilon and N. Christofides. The loading problem. *Management Science*, 17:259–268, 1971.

[19] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2(1):5–30, 1996.

[20] C.E. Ferreira, A. Martins, and R. Weismantel. Solving multiple knapsack problems by cutting planes. *SIAM Journal on Optimization*, 6(3):858–877, 1996.

[21] J.C. Fisk and M.S. Hung. A heuristic routine for solving large loading problems. *Naval Research Logistics Quarterly*, 26(4):643–650, 1979.

[22] A.S. Fukunaga. A new grouping genetic algorithm for the multiple knapsack problem. Presented at the 2008 IEEE Congress on Evolutionary Computation. Pages 2225–2232, 2008.

[23] A.S. Fukunaga. A branch-and-bound algorithm for hard multiple knapsack problems. *Annals of Operations Research*, 184(1):97–119, 2011.

[24] A.S. Fukunaga and R.E. Korf. Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of Artificial Intelligence Research*, 28:393–429, 2007.

[25] A.S. Fukunaga and S. Tazoe. Combining multiple representations in a genetic algorithm for the multiple knapsack problem. Presented at the 2009 IEEE Congress on Evolutionary Computation. Pages 2423–2430, 2009.

[26] M.R. Garey and D.S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[27] P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9:849–859, 1961.

[28] M.S. Hung and J.C. Fisk. An algorithm for 0-1 multiple-knapsack problems. *Naval Research Logistics Quarterly*, 25(3):571–579, 1978.

[29] G. Ingargiola and J.F. Korsh. An algorithm for the solution of 0-1 loading problems. *Operations Research*, 23(6):1110–1119, 1975.

[30] J.R. Kalagnanam, A.J. Davenport, and H.S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. *Electronic Commerce Research*, 1:221–238, 2001.

[31] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[32] S. Kataoka and T. Yamada. Upper and lower bounding procedures for the multiple knapsack assignment problem. *European Journal of Operational Research*, 237(2):440–447, 2014.

[33] H. Kellerer, D. Pisinger, and U. Pferschy. *Knapsack problems*. Springer, Berlin, 2004.

[34] Y. Laalaoui. Improved swap heuristic for the multiple knapsack problem. In I. Rojas, G. Joya, and J. Gabestany, editors, *Advances in Computational Intelligence: 12th International Work-Conference on Artificial Neural Networks – IWANN 2013*, pages 547–555. Springer Berlin Heidelberg, 2013.

[35] Y. Laalaoui and R. M'Hallah. A binary multiple knapsack model for single machine scheduling with machine unavailability. *Computers & Operations Research*, 72:71–82, 2016.

[36] M. Labbé, G. Laporte, and S. Martello. Upper bounds and algorithms for the maximum cardinality bin packing problem. *European Journal of Operational Research*, 149:490–498, 2003.

[37] M.E. Lalami, M. Elkihel, D. El Baz, and V. Boyer. A procedure-based heuristic for 0-1 multiple knapsack problems. *International Journal of Mathematics in Operational Research*, 4(3):214–224, 2012.

[38] S. Martello and M. Monaci. Algorithmic approaches to the multiple knapsack assignment problem. Technical report, DEI "Guglielmo Marconi", Alma Mater Studiorum Università di Bologna, Italy, 2017.

[39] S. Martello and P. Toth. Solution of the zero-one multiple knapsack problem. *European Journal of Operational Research*, 4(4):276–283, 1980.

[40] S. Martello and P. Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. *Discrete Applied Mathematics*, 3(4):275–288, 1981.

[41] S. Martello and P. Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27(2):93–112, 1981.

[42] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, 1990, (available on line at `www.or.deis.unibo.it`).

[43] J. Martinovic, G. Scheithauer, and J.M. Valério de Carvalho. A comparative study of the arcflow model and the one-cut model for one-dimensional cutting stock problems. *European Journal of Operational Research*, 266:458–471, 2018.

[44] D. Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114(3):528–541, 1999.

[45] D. Pisinger and M. Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem. *INFORMS Journal on Computing*, 19(1):36 – 51, 2007.

[46] M.R. Rao. On the cutting stock problem. *Journal of the Computer Society of India*, 7:35–39, 1976.

[47] P. Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 648–662. Springer, 2004.

[48] J. Simon, A. Apte, and E. Regnier. An application of the multiple knapsack problem: The self-sufficient marine. *European Journal of Operational Research*, 256(3):868–876, 2017.

[49] J.M. Valério de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659, 1999.

[50] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183:1109–1130, 2007.

[51] L.A. Wolsey. Valid inequalities, covering problems and discrete dynamic programs. *Annals of Discrete Mathematics*, 1:527–538, 1977.